Lecture 6 – Methods & Classes

08-671
Java for Application Programmers

January 28, 2016

Terry Lee
Assistant Teaching Professor
School of Computer Science

08-671 Lecture Topics

(subject to change – but only a little bit)

#1 Intro
#2 Primitive Types
#3 Java Classes
#4 Reference Types
#5 Loops & Arrays
#6 Methods & Classes
#7 Lists & Maps
#8 File & Network I/O
#9 Swing Interfaces
#10 Swing Actions
#11 Threads
#12 Exceptions
#13 Functional Programming
#14 In-class Written Exam

* Programming Exam – this will be a 3-hour exam

Outline

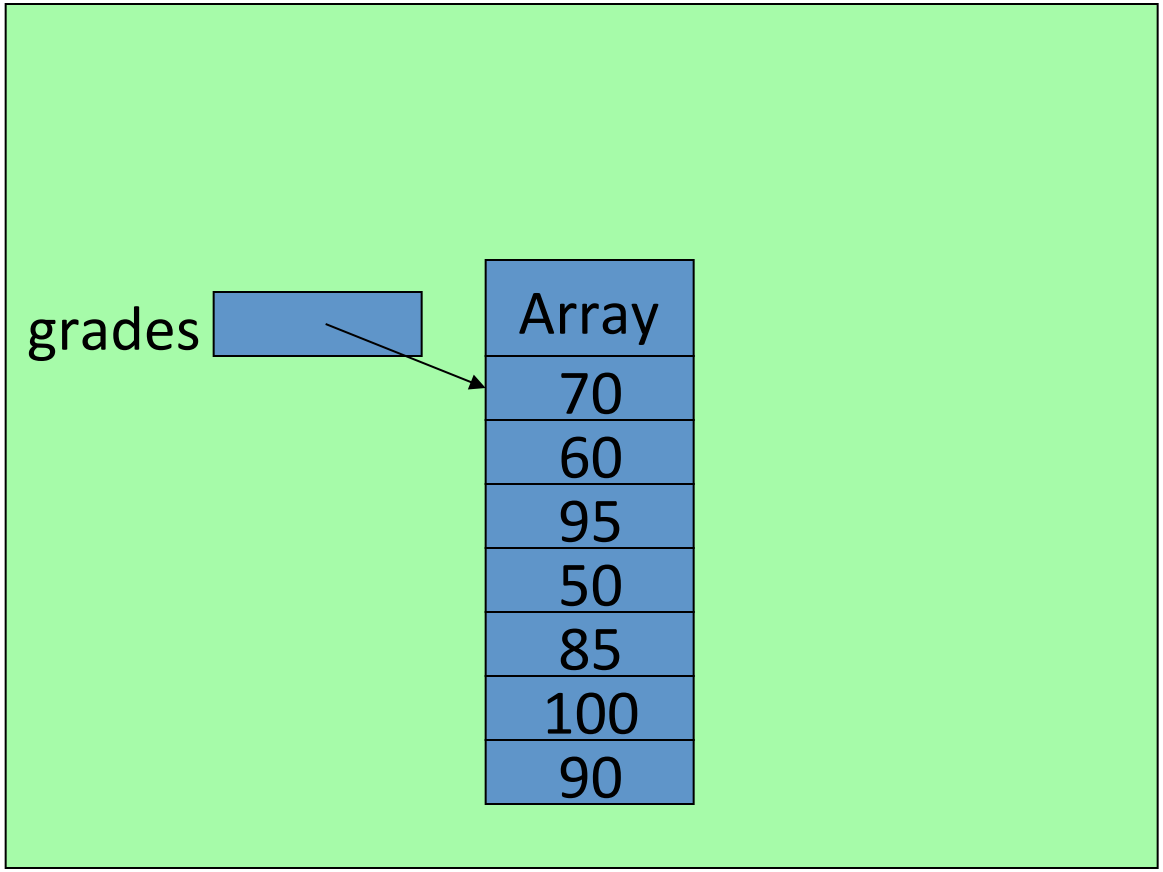
- Questions
- Methods
  - String & StringBuilder classes
- Classes
  - Static
  - Getters & Setters
  - Overriding vs. Overloading
- Packages
  - Visibility

Question for You!

- When you sort an array of **ints** do the numbers move?

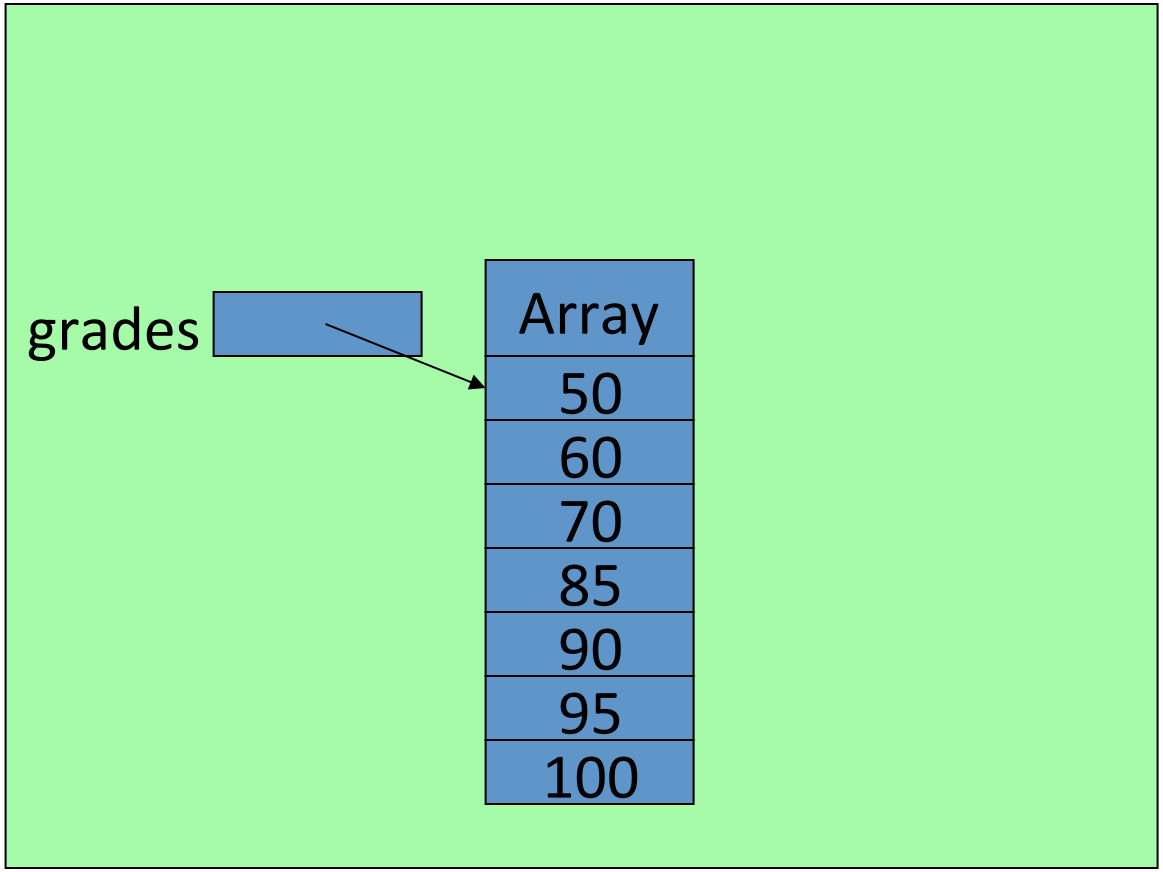
Before Sort

(sorting an array of ints in ascending order)



After Sort

(sorting an array of ints in ascending order)

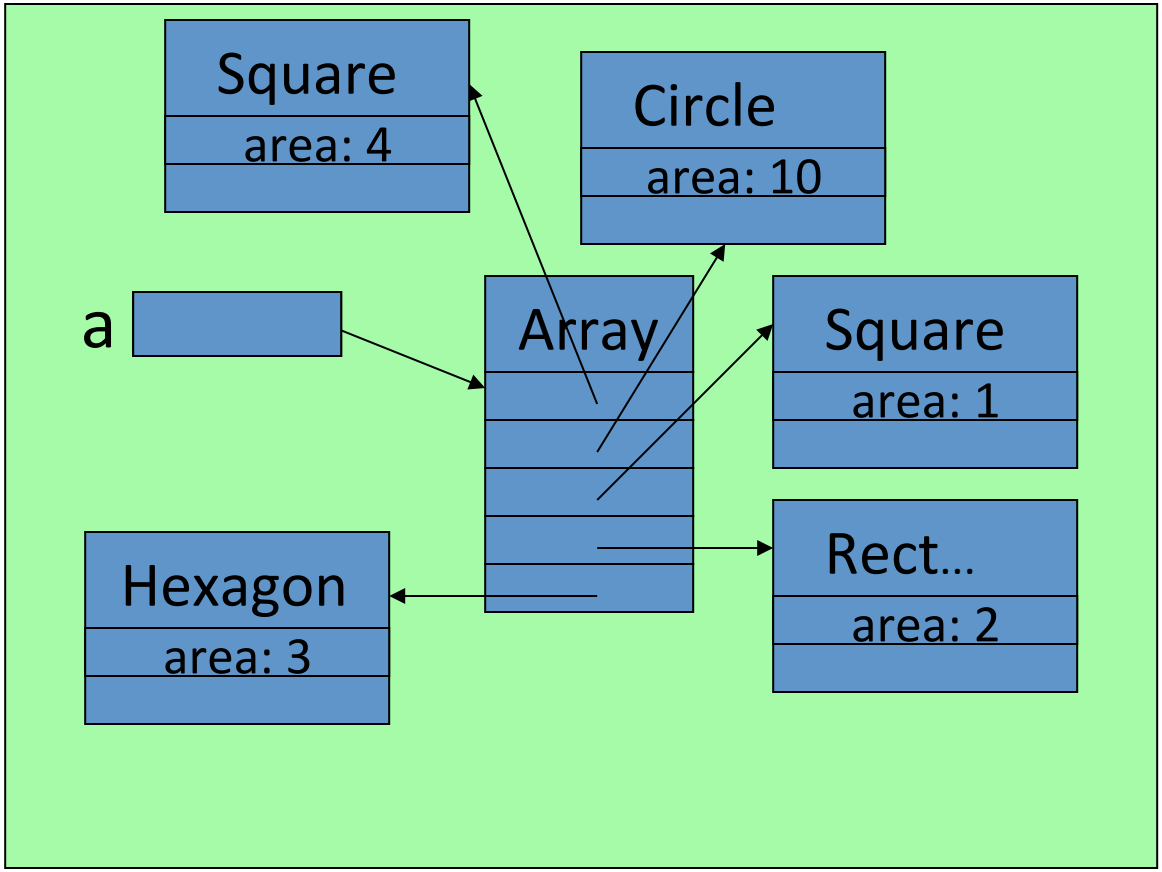


How about this?

- When you sort an array of **Objects** do the Objects move?

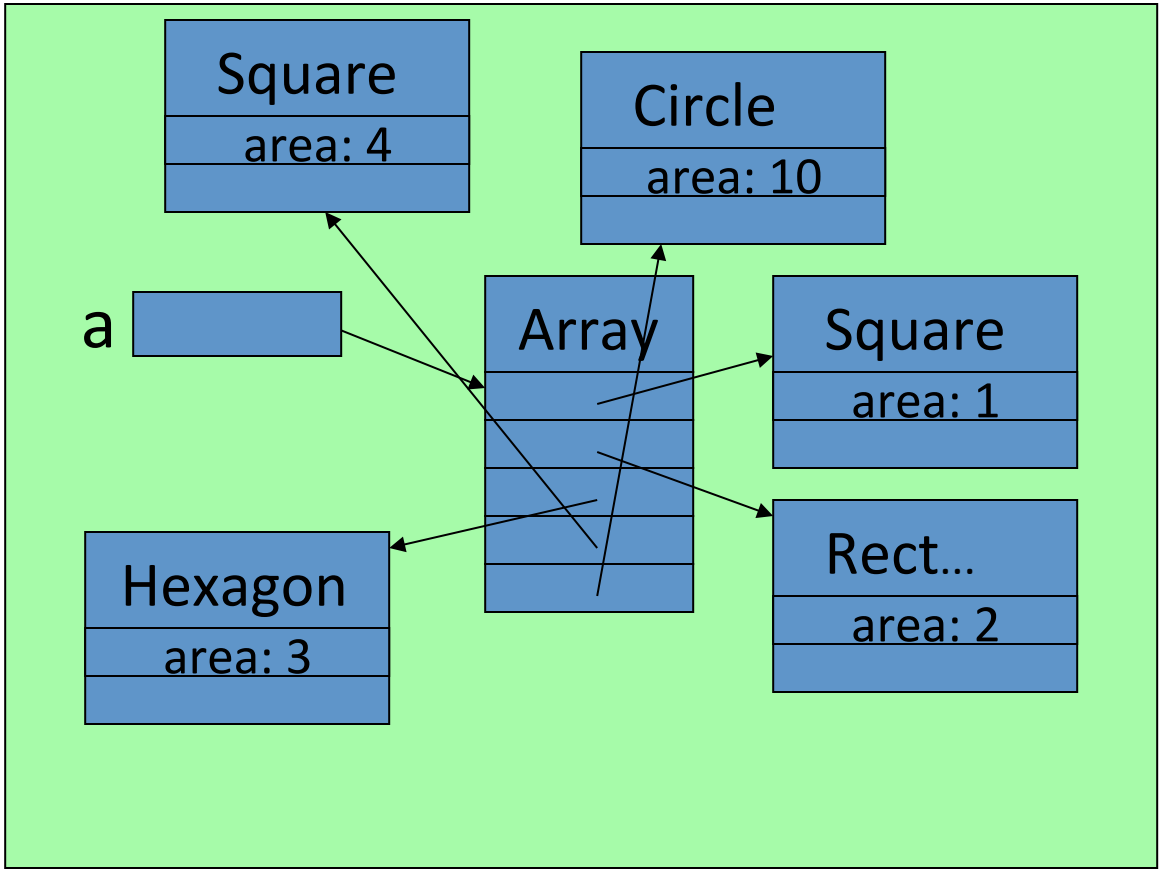
Before Sort

(sorting an array of objects in ascending order by area)



After Sort

(sorting an array of objects in ascending order by area)

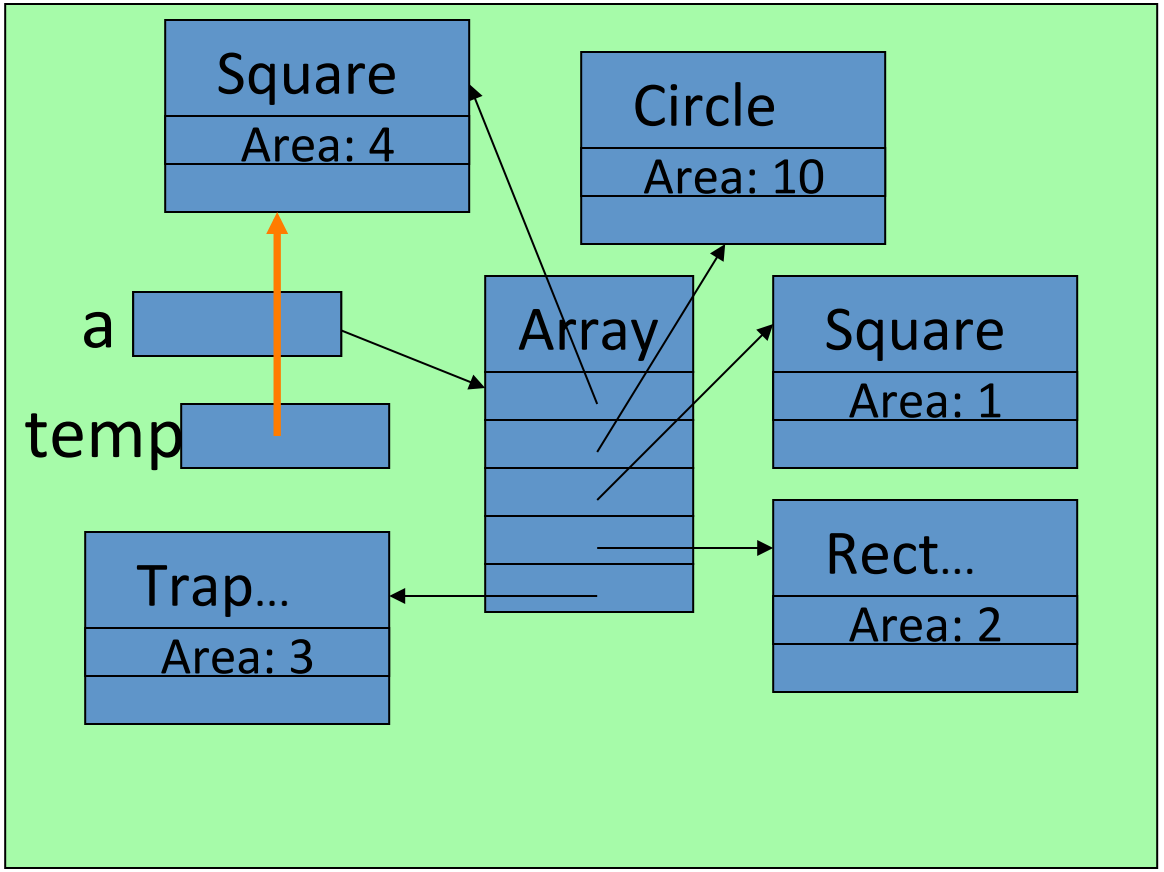


During the Sort

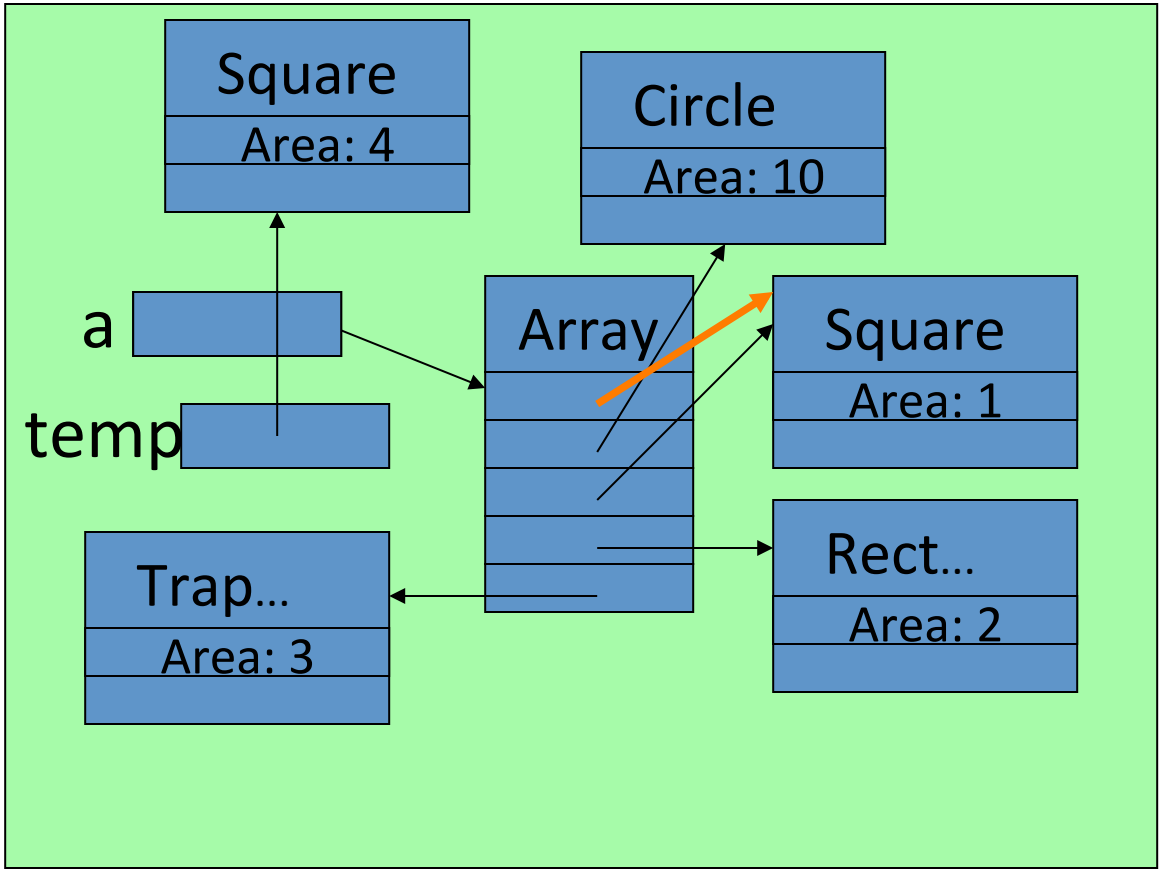
- We execute swaps to order the array:

```
temp = a[i];
a[i] = a[j];
a[j] = temp;
```

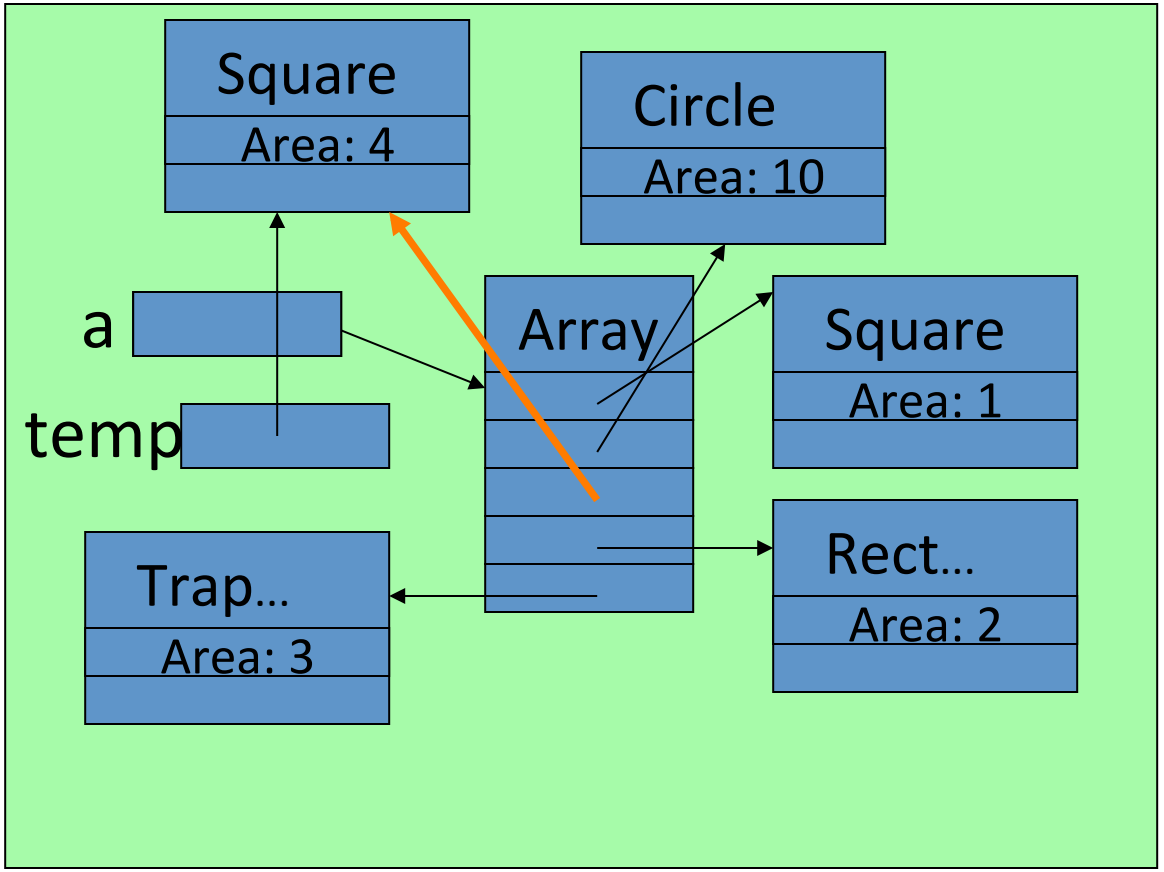
temp = a[0];



a[0] = a[2];



a[2] = temp;



Outline

- ✓ Questions
- ⇢ Methods
  - String & StringBuilder classes
- Classes
  - Static
  - Getters & Setters
  - Overriding vs. Overloading
- Packages
  - Visibility

Why Create Methods?

- Easier to understand programs
- Shorter programs
  - Long methods tend to have too many responsibilities
  - Long methods tend to be harder to understand and maintain
- Allows abstraction

Methods Allow Shorter Programs

```
public class AlicesRestaurant {
  public static void refrain() {
    System.out.println("You can get anything you want …");
    System.out.println("You can get anything you want …");
    System.out.println("Walk right in it's around the …");
    System.out.println("Just a half a mile from the ra…");
    System.out.println("You can get anything you want …");
  }

  public static void main(String args[]) {
    System.out.println("This song is called Alice's Re…");
    refrain();
    System.out.println("Now it all started about two T…");
    refrain();
    refrain();
    …
  }
}
```

Parameterization and Method Structure


```
public boolean loginOk(String user, String password) {

    // Until we learn how to access a database
    // and check this the real way…
    if (!user.equals(”terrylee")) {
        return false;
    }
    if (!password.equals(”password")) {
        return false;
    }
    return true;
}
```

*** Method name and parameters are parts of method signature**

Which is Easier to Understand?

```
public class Example1 {
    public static void main(String[] args) {
        double a = Double.parseDouble(args[0]);
        double b = Double.parseDouble(args[1]);
        double sum = a * a + b * b;

        double c = sum / 2;
        double previousC = 0;

        while (c != previousC) {
            previousC = c;
            c = (sum / c + c) / 2;
        }

        System.out.println("c = " + c);
    }
}
```

Which is Easier to Understand?

```
public class Example2 {
    public static void main(String[] args) {
        double a = Double.parseDouble(args[0]);
        double b = Double.parseDouble(args[1]);
        double c = Math.sqrt(a*a + b*b);
        System.out.println("c = " + c);
    }
}
```

Which is Easier to Understand?

```
public class Example3 {
    public static double hypotenuse(double a,
                                    double b) {
        return Math.sqrt(a*a + b*b);
    }

    public static void main(String[] args) {
        double a = Double.parseDouble(args[0]);
        double b = Double.parseDouble(args[1]);
        double c = hypotenuse(a,b);
        System.out.println("c = " + c);
    }
}
```

Return – A New Statement

- Notice the return statement
  - Exits the method. More specifically, it returns the control to its caller
  - Provide a return value
    - Return value must be provided if the declaration of the method is not void
    - Type of value returned must match the return type in the method header

Allows Abstraction

- Did you really want to know how to compute the square root?
  - Use Math.sqrt(double)
- Did you really want to think about how to implement strings?
  - Use java.lang.String (aka String)

Keep in mind…

- You are not the only one who reads your code
- You may need to read your code to debug later
- You may need to read someone else's code to debug
- So, try your best to write understandable code
  - Make your code communicate the thinking behind the code
  - Good method structure matters

Strings

- Unlike the primitive types we have studied, Strings are Java objects
  - More formally: instances of Java objects
  - (Primitive types are boolean, byte, char, double, float, int, long – note: they start with lowercase letters)
- You can't change Strings
  - You can refer to a portion of a String
  - You can copy Strings (or portions of them)
  - You can make new Strings

Documentation on Strings

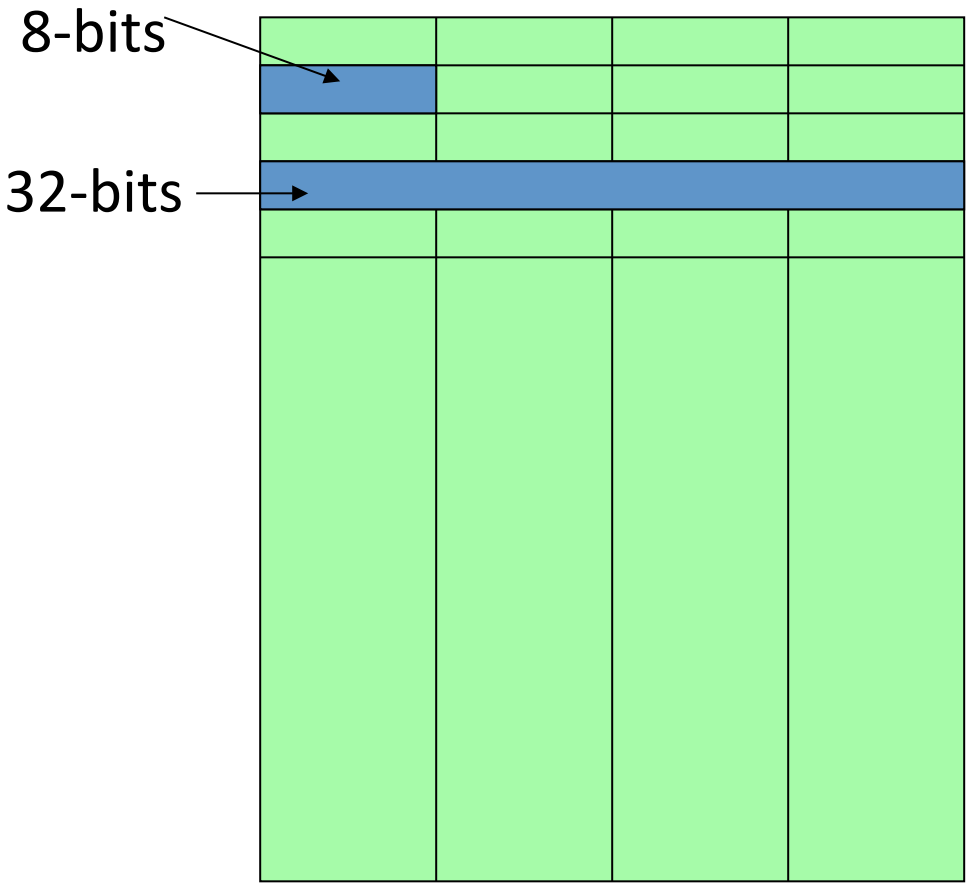
- Check on the Java Docs
- Interesting methods:
  - charAt()
  - length()
  - equals()
  - substring()

Deeper look into Strings

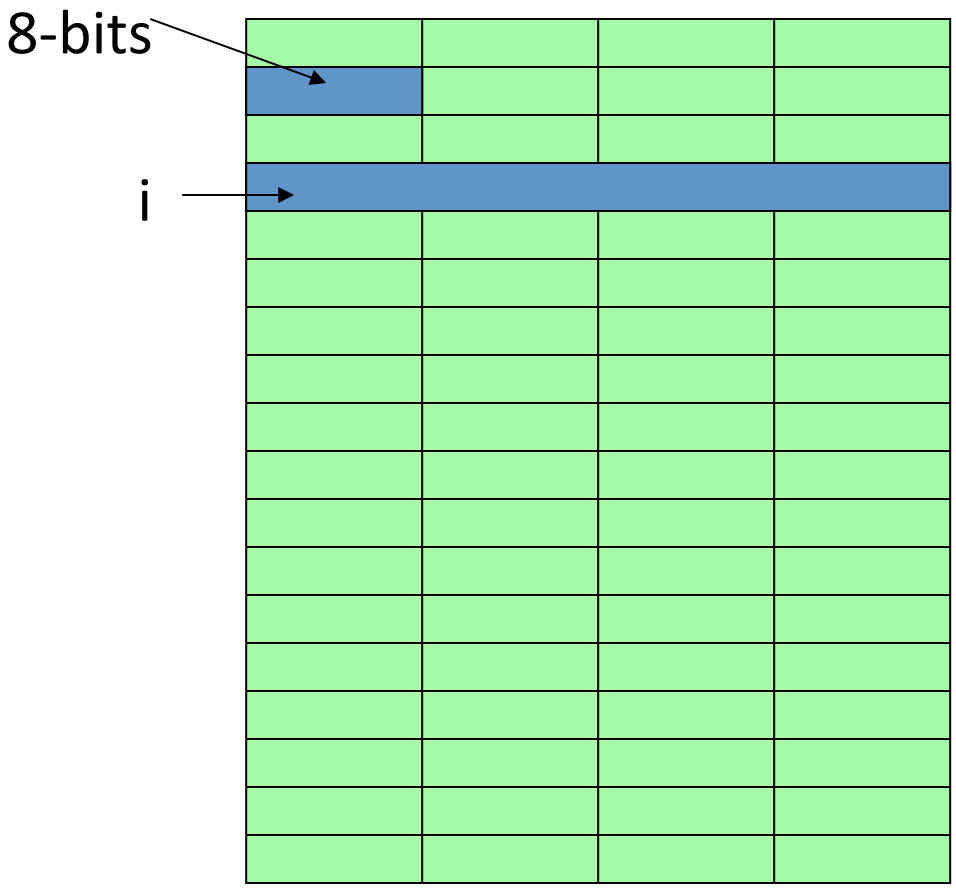
```
public class StringTest {
    public static void main(String[] args) {
        int i = 4;
        double d;
        String middle = "Lee";
        String m1 = middle;
        String m2 = args[0];

        System.out.println(i);
        System.out.println(m1);
        System.out.println(m2);
        System.out.println(m1 == middle);
        System.out.println(m1 == m2);
        System.out.println(m1.equals(m2));
    }
}
```

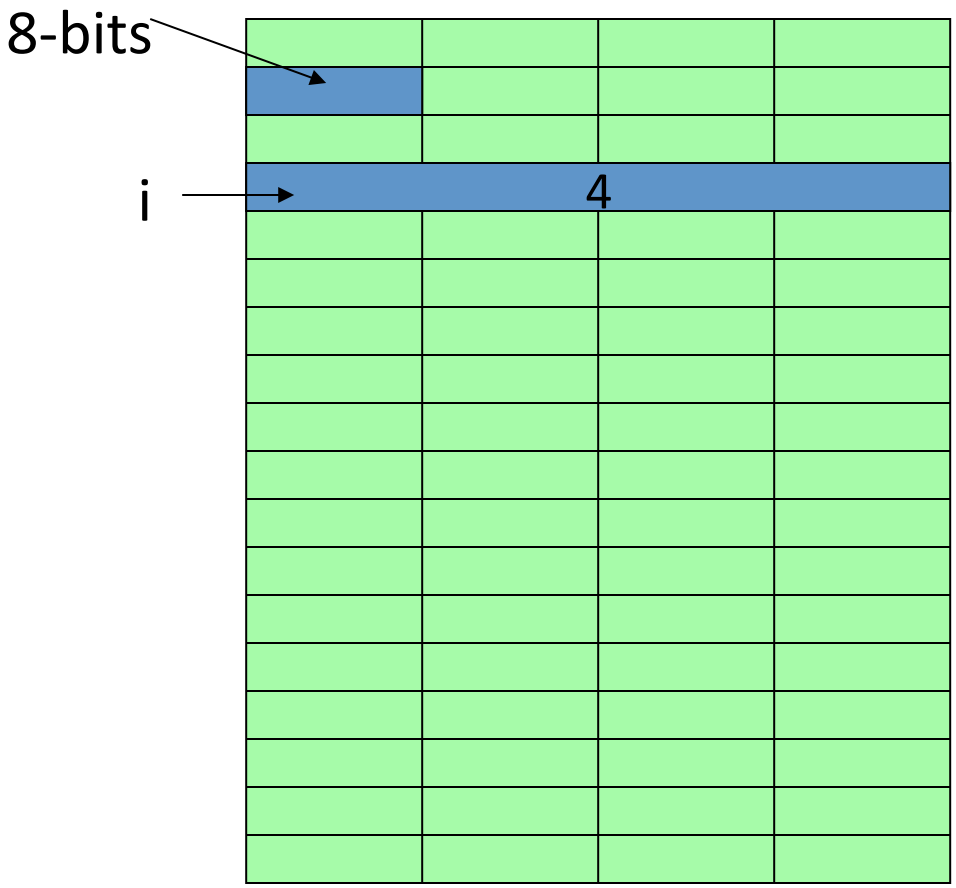
Remember Memory?



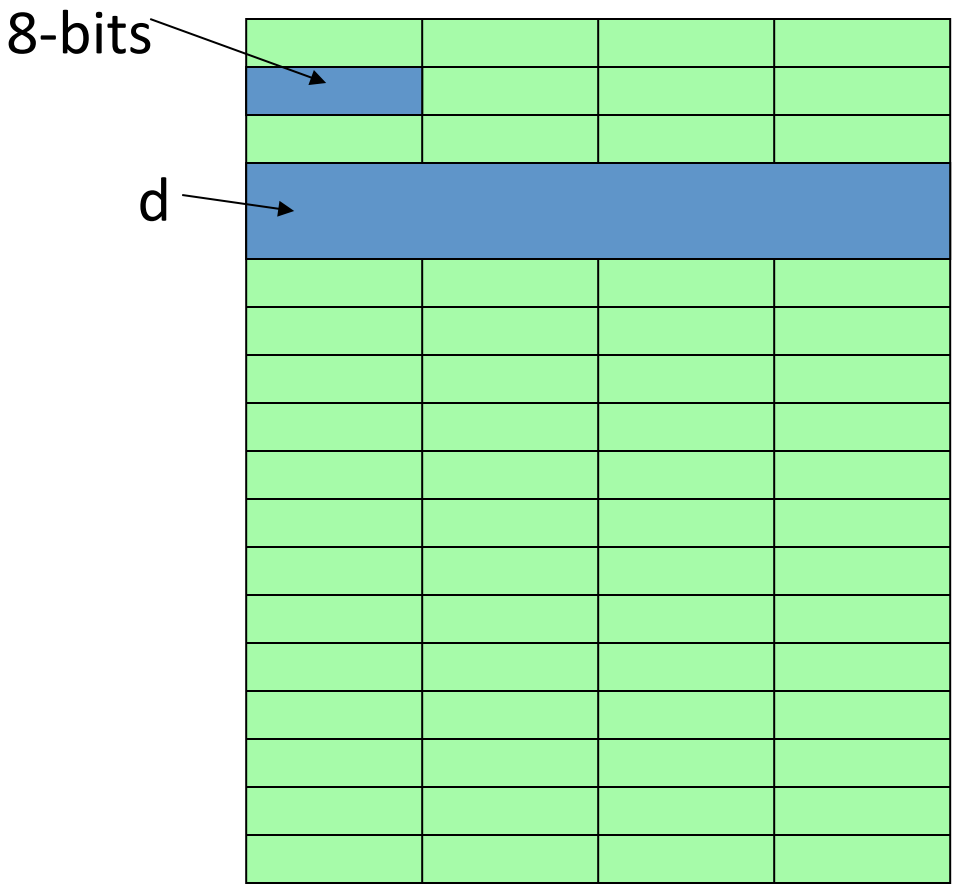
int i;



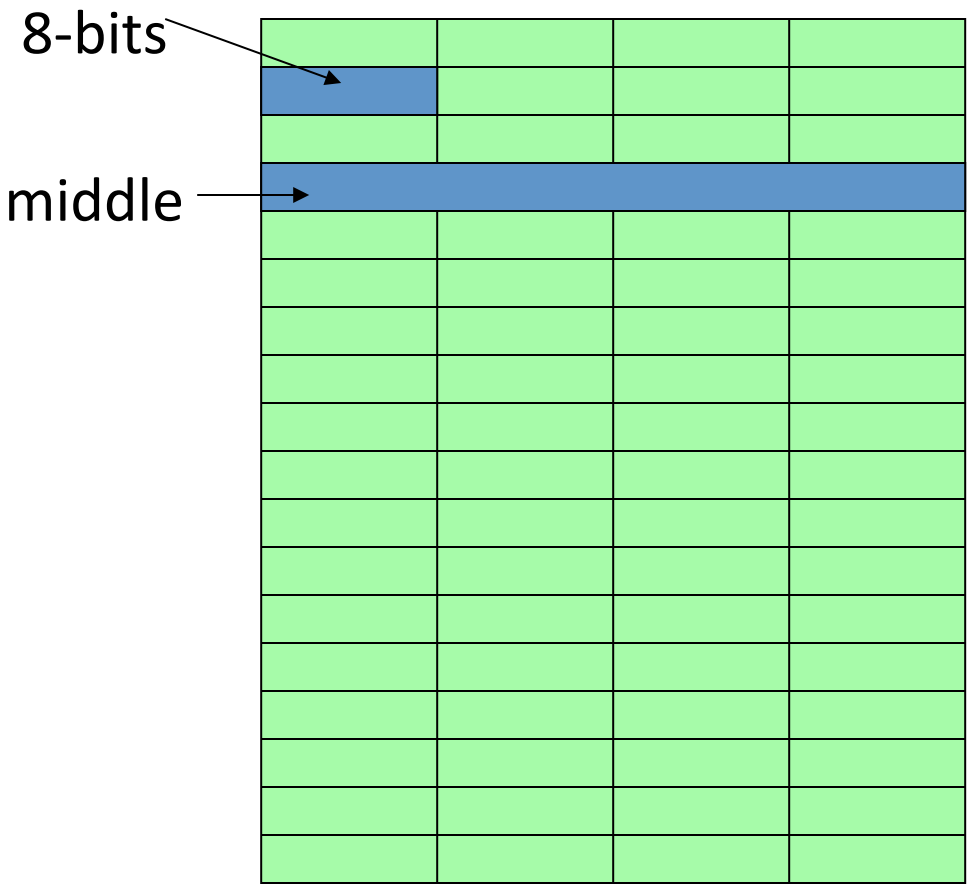
int i = 4;



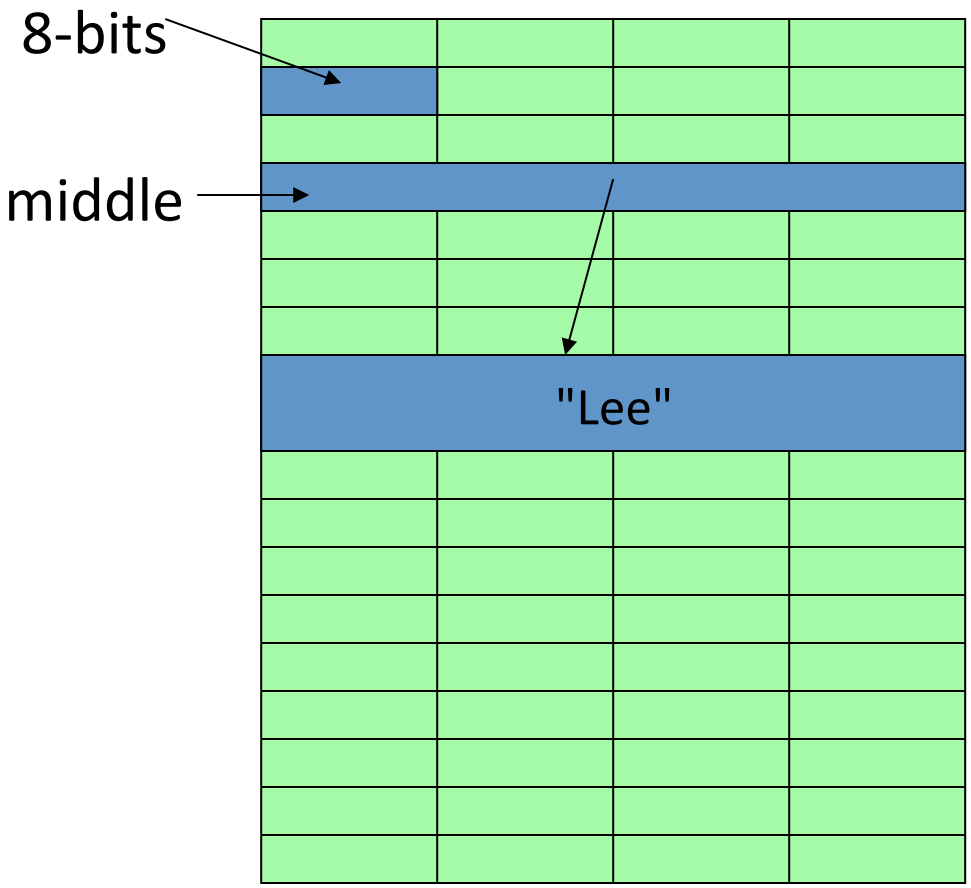
double d;



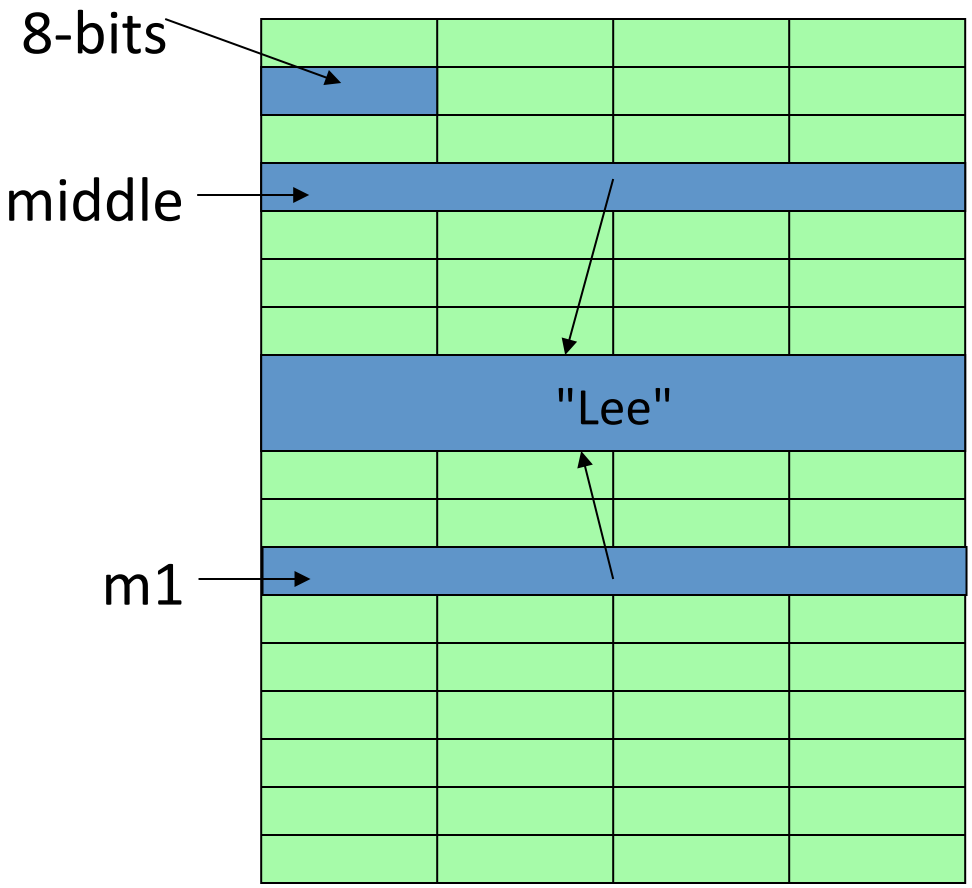
String middle;



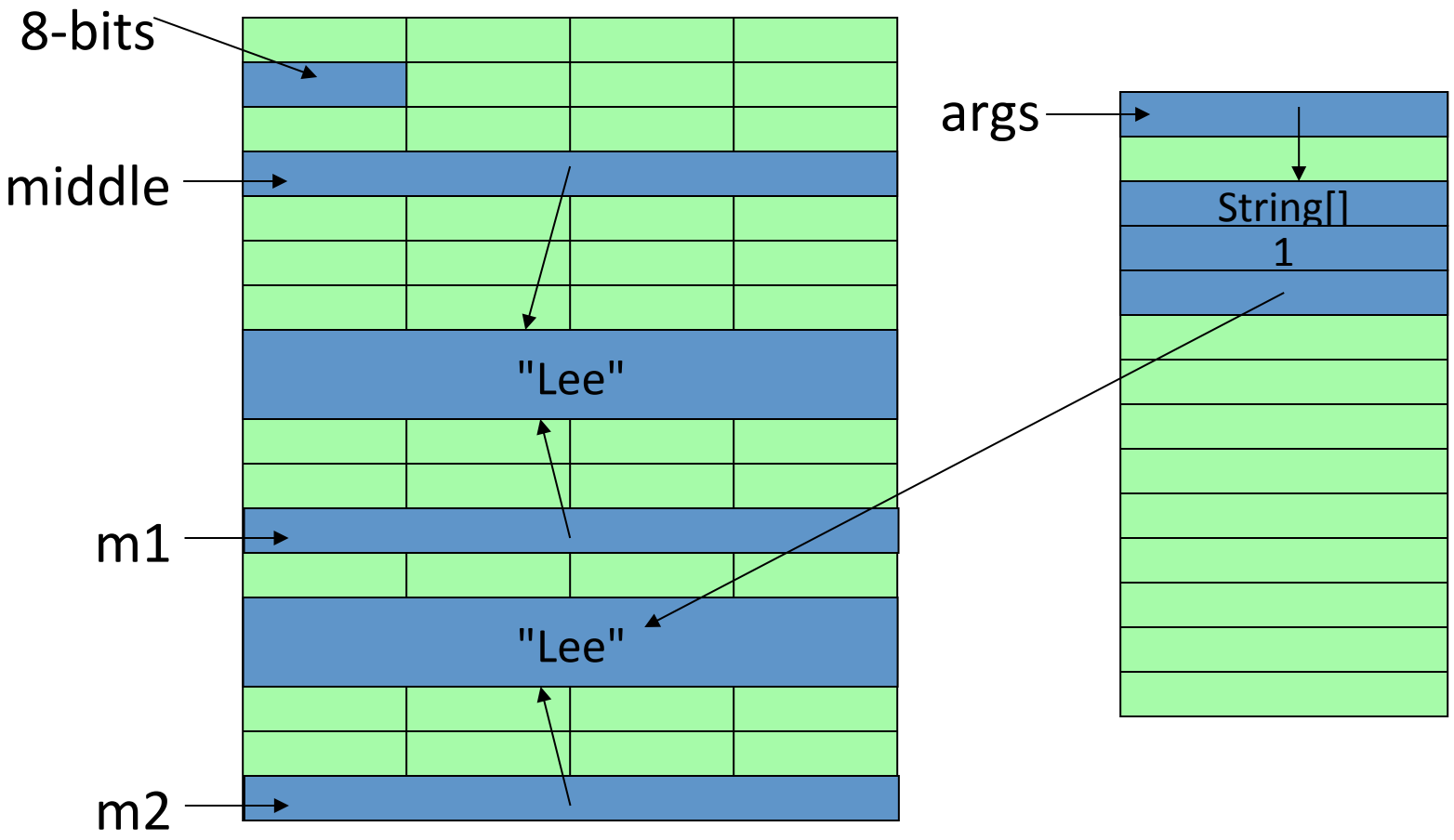
String middle = "Lee";



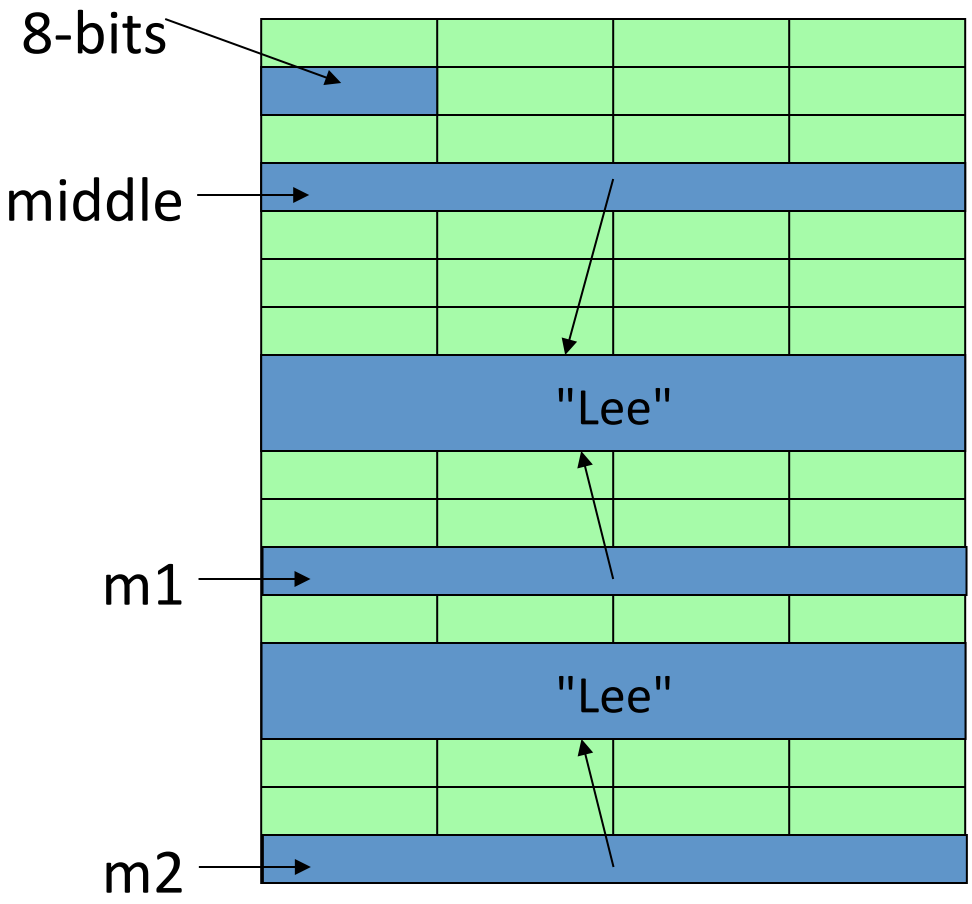
String m1 = middle;



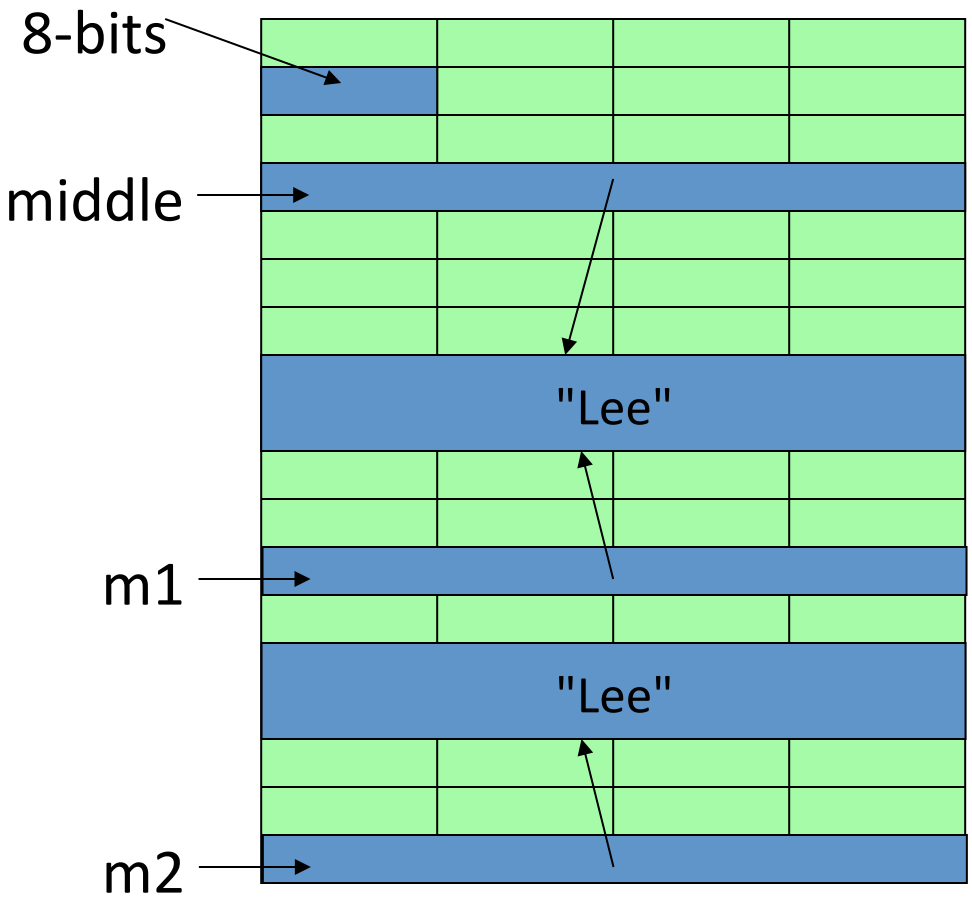
String m2 = args[0];



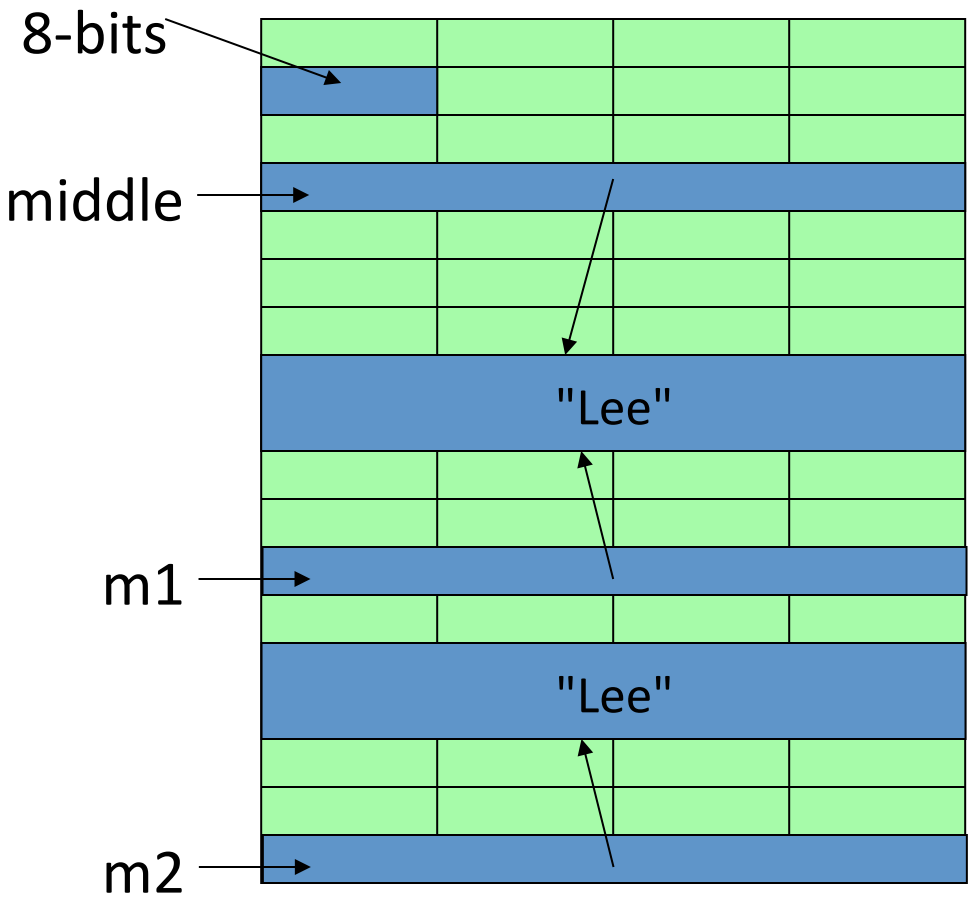
¿ m1 == middle ?



¿ m1 == m2 ?



¿ m1.equals(m2) ?



Identity vs. Equality

- "==" used to check identity
  - Whether two objects share the same memory address
- "equals" method defines equality of two objects
  - Usually checks if two instances (objects) contain the same state
  - Object class has equals method which checks identity!!!
  - When writing your own class, YOU need to think about how to define equality (in other words, **need to implement equals method**)
  - For example, equals method of String class

So this is why we use equals()

```
public boolean loginOk(String user, String password) {

    // Until we learn how to access a database
    // and check this the real way…
    if (!user.equals(”terrylee")) {
        return false;
    }
    if (!password.equals(”password")) {
        return false;
    }
    return true;
}
```

Don’t do this!

```
public class StringTest {
    public static void main(String args[]) {
        String middle = "Lee";
        String m1 = middle;
        String m2 = args[0];

        // bad way
        System.out.println(m1 == middle);
        // bad way
        System.out.println(m1 == m2);
        // good way
        System.out.println(m1.equals(m2));
    }
}
```

*** Unless your intention is to check their identity**

Strings and ints

- Converting from Strings to ints
  - Use Integer.parseInt()
- Converting from ints to Strings
  - Use String.valueOf()
  - or System.out.print()
  - or String.format() or System.out.printf()
  - or string concatenation : i+""

* Tip: Use String.valueOf() instead of concatenation

Documentation on StringBuilder

- Check on the Java Docs
- In a StringBuilder, contents can change
- Interesting methods:
  - StringBuilder()
  - append()
  - delete()
  - insert()
  - length()
  - replace()
  - reverse()
  - toString()

StringBuilder used to implement +

Given:

```
String first  = "Barack";
String middle = "Hussein";
String last   = "Obama";
```

Then:

```
String name = first + " " + middle + " " + last;
```

Is roughly equivalent to:

```
StringBuilder b = new StringBuilder();
b.append(first);
b.append(" ");
b.append(middle);
b.append(" ");
b.append(last);

String name = b.toString();
```

Note: Style of returning "this" object

- StringBuilder's append() method returns this StringBuilder object
  - The programming style allows the whole concatenation sequence with no need for a StringBuilder variable:

```
String name = new StringBuilder().append(first)
                                 .append(" ")
                                 .append(middle)
                                 .append(" ")
                                 .append(last)
                                 .toString();
```

StringBuilder Tip!

Don't do these!

```
StringBuilder().append(first + " " + middle
+ " " + last);

StringBuilder().append(first + " ")
               .append(middle + " " + last);
```

Do this!

```
StringBuilder().append(first)
               .append(" ")
               .append(middle)
               .append(" ")
               .append(last);
```

Outline

- ✓ Questions
- ✓ Methods
  - ✓ String & StringBuilder classes
- ⇢ Classes
  - Static
  - Getters & Setters
  - Overriding vs. Overloading
- Packages
  - Visibility

Class Parts

- Define the class in a file with the same name
  - Specify the superclass (or just default for Object)
- Specify the variables
  - Instance variables (a.k.a. member variables)
  - Class variables (a.k.a. static variables)
- Specify the methods
  - Constructor methods
  - Instance methods (a.k.a. member methods)
  - Class methods (a.k.a. static methods)
- When declaring variables and methods
  - Use **`static`** to make it a class variable or class method

Check out the Java Docs

- Instance methods

```
String s = new String("Lee");
s.substring(…)
s.equals(…)
```

  - Note: invoke instance methods using a reference to an **instance**
- Class (Static) methods
  - Note: invoke class methods **using the name of the class**

```
Integer.parseInt(s)
```

- Primitive wrapper classes:
  - **B**oolean, **C**haracter, **F**loat, **D**ouble, **I**nteger, **L**ong, **S**hort, **B**yte
    - Note: they all start with capital letters
  - The way to store primitive values as objects (more on this later)

    "new Integer(str).intValue()" the same as Integer.parseInt(str)
  - A collection of methods to manipulate primitive values

Static Variables and Methods

- Static variables are shared by all instances of the class!
  - Use static variables only if you want all the instances of a class to share it
  - If one instance changes the value of a static variable, all instances of the same class are affected
- Static methods CANNOT access instance members of the class
- You can access public static variables and static methods from both an instance reference and a class name
  - But, please **USE a class name** to improve readability and avoid errors

Common Errors

- Define instance variable or method that should have been static
- Define static variable or method that should have been instance

* Question to ask: Is this (variable or method) independent of any specific instance or not?

Integer.parseInt(), Math.sqrt(), Math.PI, etc.

Let's do it again! – Customer Class

```
public class Customer {
   // Class components:
   //    class variables
   //    then, class methods
   // Instance variables go here
   // Then put the methods:
   //    constructors
   //    instance methods
   // (Really can be declared in any order)
}
```

Instance Variables

```
public class Customer {
    private int    customerNumber;
    private String firstName;
    private String lastName;
}
```

*** Instance variables are unique to each instance of Customer class**

Class (Static) Variables

```
public class Customer {
    // instance variables
    private int    customerNumber;
    private String firstName;
    private String lastName;
    // class variable
    private static int lastCustNum = 0;
}
```

*** Static variables belong to Customer class, NOT to any individual instance of Customer class**

Constructor Methods

```
public class Customer {
    private int    customerNumber;
    private String firstName;
    private String lastName;
    private static int lastCustNum = 0;

    public Customer(String first, String last) {
        firstName = first;
        lastName = last;
        lastCustNum += 11;
        customerNumber = lastCustNum;
    }
}
```

*** Static variables are accessible from constructors, instance methods, and static methods**

Instance Methods

```
public class Customer {
    private int    customerNumber;
    private String firstName;
    private String lastName;
    private static int lastCustNum = 0;
    public Customer(String first, String last) {…}

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getCustomerNumber() { return customerNumber; }

    public void setFirstName(String first) { firstName = first; }
    public void setLastName(String last) { lastName = last; }

    public String toString() { … }
}
```

*** Notice there is no setter for customerNumber**

Class Methods

```
public class Customer {
    private int    customerNumber;
    private String firstName;
    private String lastName;
    private static int lastCustNum = 0;
    public Customer(String first, String last) {…}
    public String getFirstName() { return firstName; }
    public String getLastName()  { return lastName; }
    public int getCustomerNumber() { return customerNumber; }
    public void setFirstName(String first) { firstName = first; }
    public void setLastName(String last) { lastName = last; }
    public String toString() { … }

    // static method
    public static int getNumCustomers() {
        return lastCustNum / 11;
    }
}
```

The Usual Ordering

```
public class Customer {
    // class (static) variables and methods first
    private static int lastCustNum = 0;
    public static int getNumCustomers() {
        return lastCustNum / 11;
    }

    private int    customerNumber;
    private String firstName;
    private String lastName;

    public Customer(String first, String last) {…}

    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public int getCustomerNumber(){ return customerNumber; }
    public void setFirstName(String first) { firstName = first; }
    public void setLastName(String last) { lastName = last; }
    public String toString() { … }
}
```

Getters & Setters (Encapsulation)

- (a.k.a. Accessors and Mutators)
- Fields are private
- The getter methods provide read access
- The setters provide write access
  - No setter method => makes something read-only
  - Immutable objects have no setter methods
    - Strings are immutable
    - Dates should be immutable…more on that later

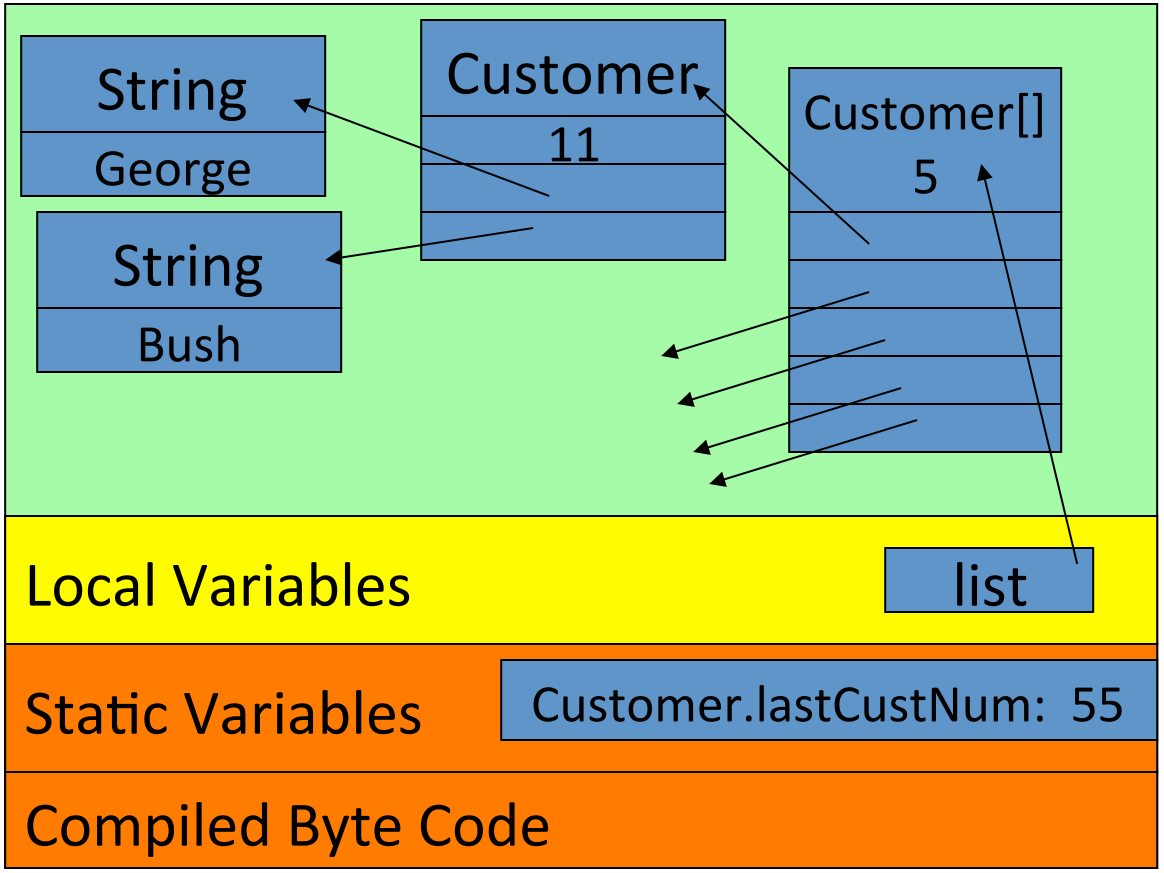
Examples

- Customer.java
- CustomerTest.java

* Notice that it is NOT necessary for an instance of the class to be created to execute static methods. Simply use Classname to execute

```
Customer.getNumCustomers()
```

How Does It **Really, Really** Work?



Overloading vs. Overriding

- "Overloading" is when you have many methods with the same name, but different parameter lists which means different signatures
  - E.g., System.out.println(…)
  - E.g., StringBuilder.append(…)
- "Overriding" is when you replace a superclass's method in a subclass. (Same signatures)
  - E.g., toString()

* Note (one more time): method signature consists of its name and parameter lists. **The method's return type is not a part of the signature.**

Outline

- ✓ Questions
- ✓ Methods
  - ✓ String & StringBuilder classes
- ✓ Classes
  - ✓ Static
  - ✓ Getters & Setters
  - ✓ Overriding vs. Overloading
- ⇢ Packages
  - – Visibility

Java Package

- A package is a collection of related classes
  - Main reason is to guarantee uniqueness of class names
- We've been using the java.lang package
  - Automatically or implicitly imported already
  - Do not need to write import for any class in java.lang package
- There are many others
  - Example: java.util and java.io
- To use a class from a package, you must either:
  - Specify the package when using the class (tedious)
  - Import the class using an import statement
  - Import the entire package using import * (wildcard import)
  - bytecodes in class files are the same (using full name) because import statement simply tells the compiler where to locate the class

Recall Our Selection Sort

```
for (int i = 0; i < grades.length; i++) {
    for (int j = i + 1; j < grades.length; j++) {
        if (grades[j] < grades[i]) {
            int temp = grades[i];
            grades[i] = grades[j];
            grades[j] = temp;
        }
    }
}
```

Check out java.util.Arrays

- Notice all methods are static methods
- There are sort methods!!

Replace with...

Full name (tedious)

```
java.util.Arrays.sort(grades)
```

or (explicit import, preferred)

```
import java.util.Arrays;
…
Arrays.sort(grades);
```

or (wildcard import)

```
import java.util.*;
…
Arrays.sort(grades);
```

Check it out…

- Examples
  - SortGrades.java
  - SortGradesEasy.java

Can it work for Strings?

```
for (int i = 0; i < names.length; i++) {
    for (int j = i + 1; j < names.length; j++) {
        if (names[j] < names[i]) {
            String temp = names[i];
            names[i] = names[j];
            names[j] = temp;
        }
    }
}
```

You Must Use compareTo()

```
for (int i = 0; i < names.length; i++) {
    for (int j = i + 1; j < names.length; j++) {
        if (names[i].compareTo(names[j]) > 0) {
            String temp = names[i];
            names[i] = names[j];
            names[j] = temp;
        }
    }
}
```

*** Remember this. This is one of the common mistakes!**

More Examples of Sorting

- SortStrings.java
- SortStringsEasy.java
- **`java.util.Arrays.sort()`** can sort objects that implement compareTo()

* Just like equals method, need to implement compareTo() method in your class which ought to define natural ordering of the class

Haha….

- You can implement compareTo() for your Shape class, but…
  - It will either sort by area or perimeter
- So you cannot use Arrays.sort() for both sorts

`java.util.Comparator`

- Pass a Comparator to sort to provide an alternative ordering

… But we won't get to this until next week.

So, you may stick with the sorting algorithm

Visibility (Access) Modifiers

- Use **`public`** to make it accessible outside the class
- Use **`private`** to make it accessible only from within the class
- Use **`protected`** to make it accessible from subclasses
  - Subclasses in any package or classes in the same package
- (Say nothing, you get **default** package-private accessibility)
  - Any class in the same package

Visibility Modifiers

Visibility increases

private, default (no modifier), protected, public

Advanced Topics in Classes

- Later we will cover
  - Abstract classes
  - Anonymous classes
  - Nested classes (non-static vs. static)
  - Enumeration classes

Next Week

- Lists & Maps (a little about Data Structures)
  - Head First Java Chapter 16
- File I/O & Network I/O
  - Head First Java Chapters 14 and 15