Object Oriented Programming

1. Classes and Objects
2. Arrays

Methods: How call works

```
// Author : Fred Swartz
import javax.swing.*;
public class KmToMiles {
   private static double convertKmToMi(double kilometers) {
      return kilometers * 0.621; // There are 0.621 miles in a kilometer.
   }
   public static void main(String[] args) {
      //... Local variables
      String kmStr;    // String km before conversion to double.
      double km;       // Number of kilometers.
      double mi;       // Number of miles.
      //... Input
      kmStr = JOptionPane.showInputDialog(null, "Enter kilometers.");
      km = Double.parseDouble(kmStr);
      //... Computation
      mi = convertKmToMi(km) ;
      //... Output
      JOptionPane.showMessageDialog(null, km + " kilometers is " + mi + "
        miles.");
   }
}
```

Methods: How call works

- Consider `mi = convertKmToMi(km);`
- The steps to process this statement are:
  1. Evaluate the arguments left-to-right
  2. Push a new stack frame on the call stack. Parameter and local variable storage (parameter kilometers only, here). The saved state of the calling method (includes calling method return address)
  3. Initialize the parameters. When the arguments are evaluated, they are assigned to the local parameters in the called method.
  4. Execute the method
  5. Return from the method. The stack frame storage for the called method is popped off the call stack

Creating Objects

- The Java language has three mechanisms dedicated to ensuring proper initialization of objects:
  - *instance initializers* (also called instance initialization blocks),
  - *instance variable initializers*, and
  - *constructors*.
    - Note. Instance initializers and instance variable initializers collectively are called "initializers."
- All three mechanisms => code that is executed automatically when an object is created.
- When you allocate memory for a new object with the new operator or the newInstance() method of class Class, the Java virtual machine will insure that initialization code is run before you can use the newly-allocated memory.

Creating Objects

- When the **`new`** operator is invoked:
  - Memory allocation occurs (space for new object is reserved). Instance variables initialized to their default values
  - Explicit initialization performed. Variables initialized in the attribute declaration get declared values
  - A constructor is executed. Variable values may be changed by constructor
  - The reference to the object is assigned to the variable
- Example:

```
Car beetle = new Car("Volskwagen Beetle",
  Color.orange, 80, 160, 10);
```

Default Initial Values for Fields

Type	Value
boolean	false
byte	(byte) 0
short	(short) 0
int	0
long	0L
char	\u0000
float	0.0f
double	0.0d
object reference	null

Initializing Fields

- Simple assignment
  - If we can provide an initial value for a field in its declaration.
  - E.g.
    - public static int capacity = 10; //initialize to 10
    - private boolean full = false; //initialize to false
- Static initialization blocks
  - normal block of code enclosed in braces, { }, and preceded by the static keyword
  - E.g.
    - static {

    // whatever code is needed for initialization goes here

    }
  - Used for initialization of class variables
  - static initialization blocks are called in the order that they appear in the source code

Initializing Fields

- Alternative to static blocks: private static method:
- E.g.
  - class Whatever {

    public static varType myVar = initializeClassVariable();

    private static varType initializeClassVariable() { //initialization code goes here

    }

  }

Initializing Fields

- Initializing Instance Members
  - Similar to static initialization blocks but not static
  - Java compiler copies initializer blocks into every constructor
  - E.g.

```
{
  // whatever code is needed for initialization goes here
}
```

Initializing Fields

- Initializing Instance Members
  - using a **final** method for initializing an instance variable:

```
class Whatever {
    private varType myVar = initializeInstanceVariable();
    protected final varType initializeInstanceVariable()
    { //initialization code goes here }
}
```

  - Especially useful if subclasses might want to reuse the initialization method.
  - The method is **final** because calling non-final methods during instance initialization can cause problems

Creating Objects

- When a constructor is called
  - Storage space is allocated *on the heap* for the object
  - *Each object gets its own space (own copy of the instance variables)*
  - The objects state is initialized according to the (programmer-defined) code for the class
  - We will come back to this when we discuss inheritance, to see the details there
- Note that declaring a variable of an object type produces a reference to the object, but not the object itself. To get the object itself use **`new`** and a constructor for the class

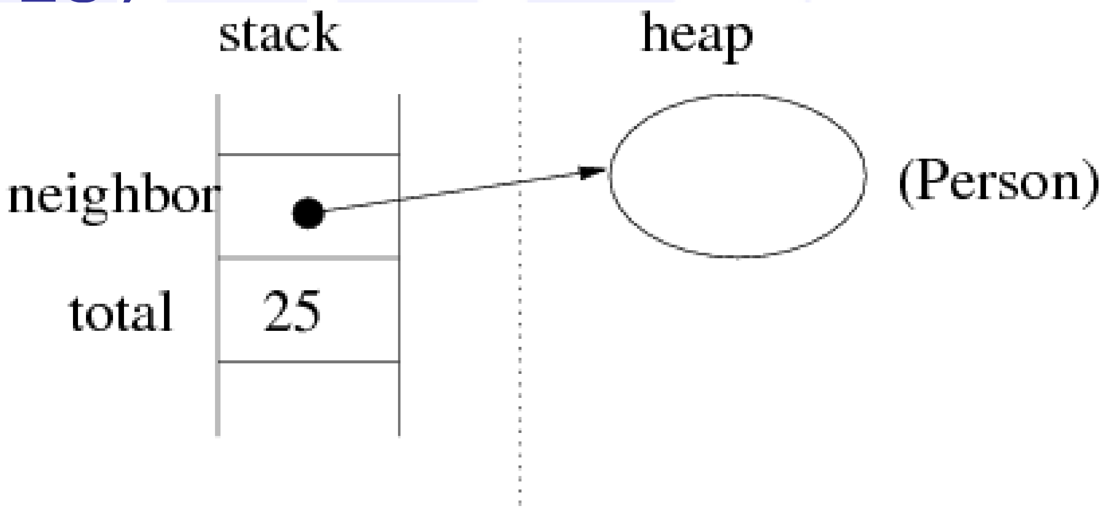
Creating Objects

- You can combine the declaration and the initialization

```
Person neighbor = new Person();
```

- just as you can for primitive types

```
int total = 25;
```



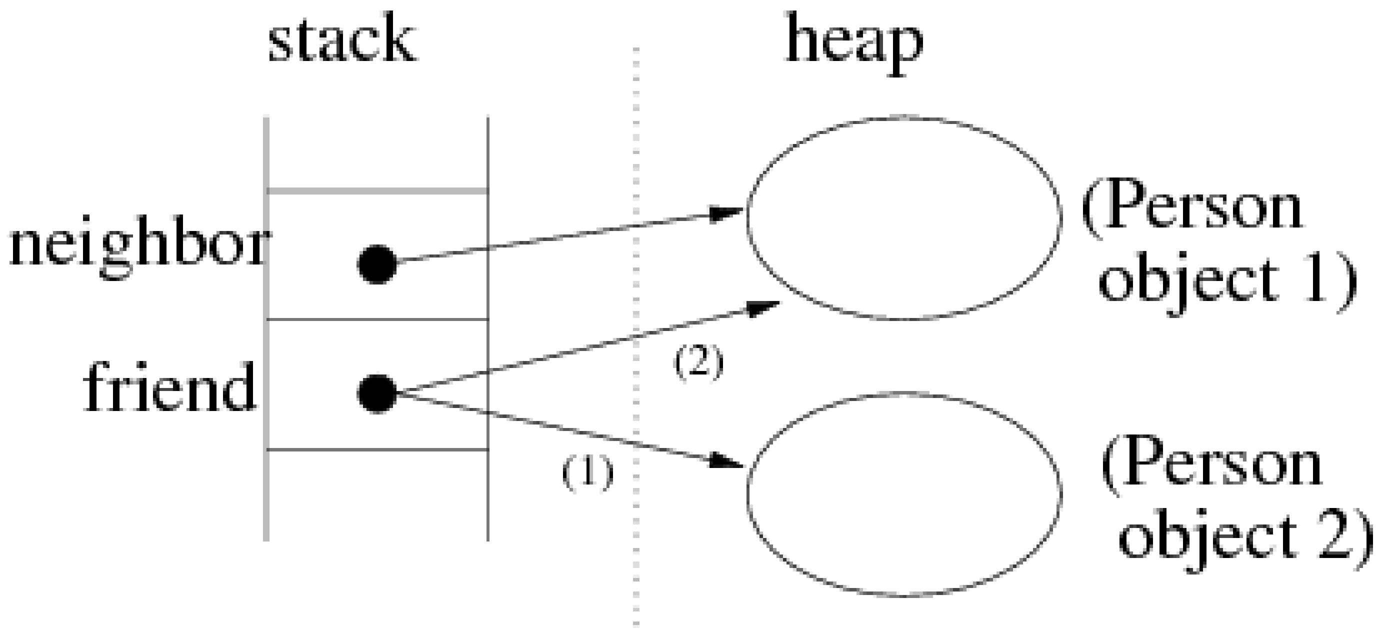
Object Assignment and Aliases

- The meaning of assignment is *different* for objects than is for primitive types

```
int num1 = 5;
int num2 = 12;
num2 = num1; // num2 holds 5 now
//---------------------------------------------
Person neighbor = new Person(); // creates object1
Person friend = new Person(); // creates object2
friend = neighbor;
```

- At the end both `friend` and `neighbor` refer to object1 (they are **aliases** of each other) and nothing refers to object2 (it is **inaccessible**)

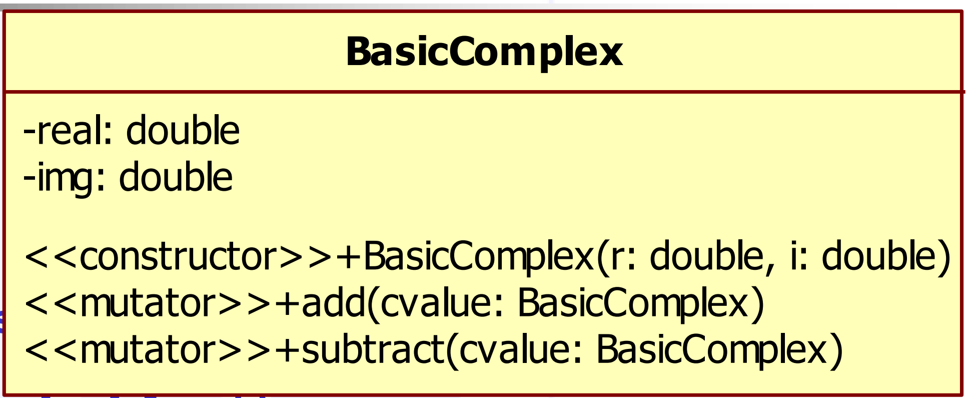
Object Assignment and Aliases



- Java will *automatically garbage collect* object2

A Very Simple "Complex" Class

```
public class BasicComplex
{
   //Properties
   private double real;
   private double img;
   // Constructor that initializes
   // the values
   public BasicComplex (double r, double i)
   { real = r; img = i; }
   // Define an add method
   public void add (BasicComplex cvalue) {
      real = real + cvalue.real;
      img  = img  + cvalue.img;
   }
   // Define a subtract method
   public void subtract (BasicComplex cvalue) {
      real = real - cvalue.real;
      img  = img  - cvalue.img;
   }
}
```


BasicComplex
-real: double
-img: double
<<constructor>>+BasicComplex(r: double, i: double)
<<mutator>>+add(cvalue: BasicComplex)
<<mutator>>+subtract(cvalue: BasicComplex)

Note. The code on the slides does not fully obey the style rules due to little space. Your code must obey the style rules, as you do not have such limitations

A Simple Switch Class

```
// A simple on/off switch (not attached to
// anything).
class SimpleSwitch {
   // Turn the switch on.
   public void switchOn(){
      System.out.println("Turn the switch on.");
      setOn(true);
   }
   // Turn the switch off.
   public void switchOff(){
      System.out.println("Turn the switch off.");
      setOn(false);
   }
   // Tell an enquirer whether the switch is on
   //  or not.
   public boolean isTheSwitchOn(){
     return getOn();
   }
// Return the state of the switch.
   private boolean getOn(){     return on;   }
   // Set the state of the switch.
   private void setOn(boolean o){
      on = o;
   }
   // Whether the switch is on or not.
   // true means on and false means off.
   private boolean on = false;
}
```

- Note that java interface documentation will NOT be properly generated for this for of writing

A Tic Tac Toe Game

```
public class TicTacToe{
  //Instance variables
  /* 2D array of chars for the board
   */
   private char[][] board;

  /** Constructor - create a board where each
    * square holds the underline '_' character.
    */
   public TicTacToe()
  {
     board = new char[3][3];
     for (int row = 0; row < 3; row ++)
    {
      for (int col =  0; col < 3; col++)
     {
        board[row][col] = '_';
     } // end of inner loop
    } // end of outer loop
  }
```

```
/** Puts character c at the
  * board's [row][col]  position if row, col,
  * and c  are valid and the square at
  * [row][col] has not already been
  * assigned a value (other than the default '_').
  * @param row board row
  * @param col board column
  * @param c character used to mark
  * @return  true if successful, false otherwise.
  */
public boolean set(int row, int col, char c)
{
  if (row >= 3 || row < 0)
    return false;
  if (col >= 3 || col < 0)
    return false;
  if (board[row][col] != '_')
    return false;
  if ( !(c == 'X' || c == 'O'))
    return false;
  // assertion: row, col, c are valid
  board[row][col] = c;
  return true;
}
```

A Tic Tac Toe Game (cont'd)

```
  /**
   * @return character at the board's [row][col] position.
   * @param row board row
   * @param col board col
   */
  public char get(int row, int col)
  {
   return board[row][col];
  }
 /** Prints the state of the board, e.g.
   *    _ _ _
   *    _ X O
   *    O _ X
   */
  public void print(){
   for (int row = 0; row < 3; row ++){
    for (int col =  0; col < 3; col++){
     System.out.print(board[row][col] + " ");
    } // end of inner loop
    System.out.println();
   } // end of outer loop
  }
 }
```

Exercises:

- complete the game to make playable (maybe by defining a new class?)
- change to accommodate larger board sizes

Class Design Hints

- Always keep data *private*
  - changes in data representation do not affect the user of the class; bugs easier to detect
- Always *initialize* data
  - Java won't initialize local variables, but it will initialize instance variables of objects. Don't rely on defaults, but initialize variables explicitly
- Don't use *too many types* in a class
  - Replace multiple *related* uses of basic types with other classes. E.g.

```
private String street;
private String city;
private String state;
private String country;
private int zip;
```

⇨

```
class Address {
   private String street;
   private String city;
   private String state;
   private String country;
   private int zip;
   . . .
}
```

Class Design Hints

- Not all fields need individual field accessors and mutators
  - E.g. Employee – get and set salary but not hiring date (it. doesn't change once created)
- Use a standard form for class definition, e.g.
  - public features
  - package scope features
  - private features

Class Design Hints

- Use a standard form for class definition, and for each section list in order
  - constants
  - constructors
  - methods
  - static methods
  - instance variables
  - static variables
- Why this? Users are more interested in the public interface than private data; also more interested in methods than data

Class Design Hints

- Break up classes with too many responsibilities, e.g.

```
class CardDeck { // bad design
  public void CardDeck() { . . . }
  public void shuffle() { . . . }
  public void getTopValue() { . . . }
  public void getTopSuit() { . . . }
  public void topRank() { . . . }
  public void draw() { . . . }

  private int[] value;
  private int[] suit;
  private int cards;
}
// create Card class!
```

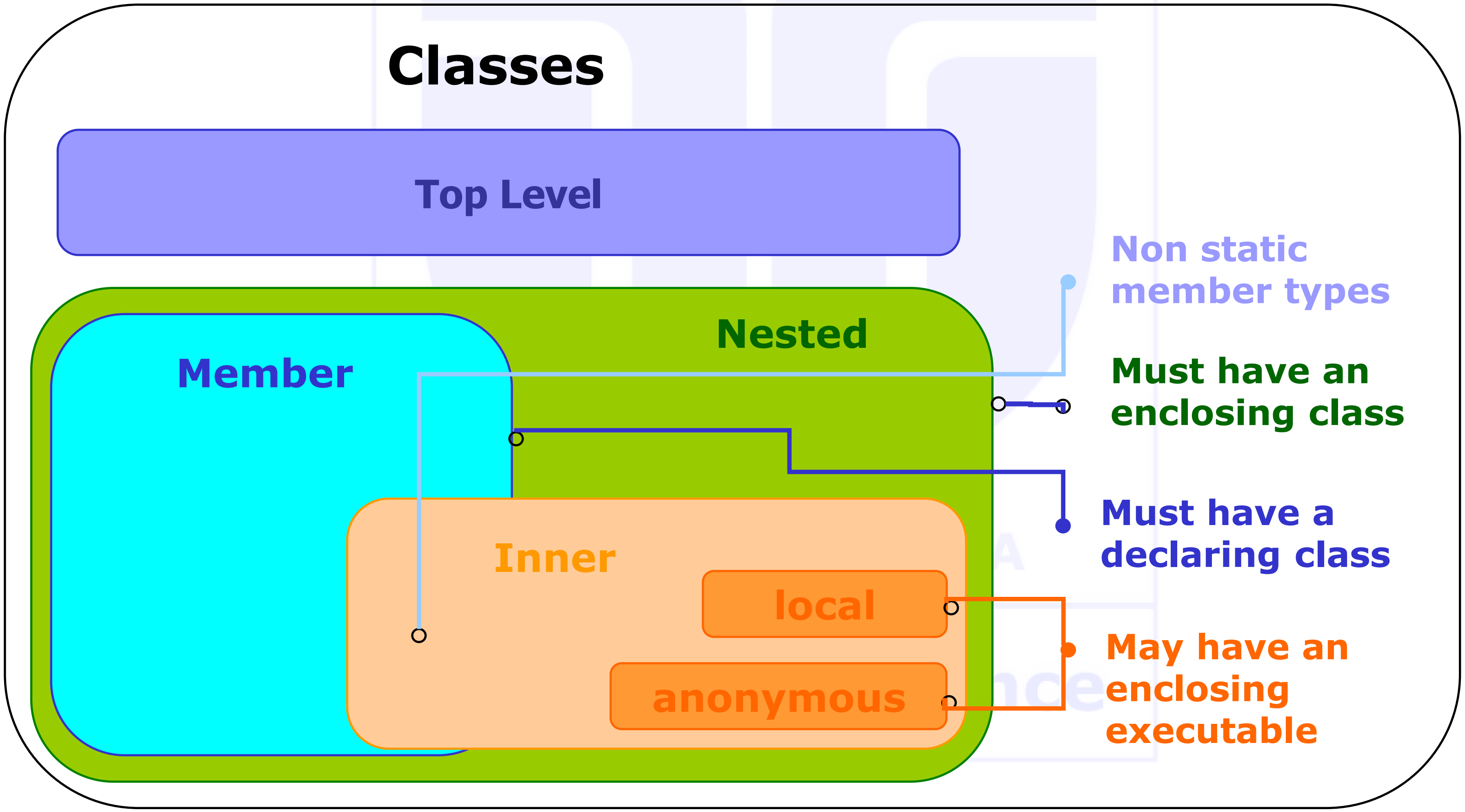
Class Design Hints

- Make the names of classes and methods reflect their responsibilities
  - Good convention:
    - Class name: noun (e.g. Order) or adjective+noun (e.g. RushOrder) or gerund+noun (e.g. BillingOrder)
    - Method names: verbs; accessors begin with "get"; mutators begin with "set"

Kinds of classes in Java

- A *top level class* does not appear inside another class or interface
- If a type is not top level it is *nested*
  - a type can be a *member* of another type; a member type is directly enclosed by another type declaration
    - some member types are *inner classes* and include:
      - *local classes*, which are named classes declared inside of a block like a method or constructor body
      - *anonymous classes*, which are unnamed classes whose instances are created in expressions and statements

Kinds of Classes in Java


Classes
Top Level
Nested
Member
Inner
local
anonymous
Non static member types
Must have an enclosing class
Must have a declaring class
May have an enclosing executable

Inner Classes

- A class definition within another class definition is called an *inner class*
  - it allows you to *group classes* that logically belong together and to *control the visibility* of one within the other
  - inner classes are distinctly different from composition
- To create an object of the inner class anywhere except from within a non-**static** method of the outer class, you must specify the type of that object as *OuterClassName.InnerClassName*

Inner Classes Example

- Typically, an outer class will have a method that returns a reference to an inner class

```
public class Parcel {
  class Contents {
    private int val = 10;
    public int value() { return val; }
  }
  class Destination {
    private String label;
    Destination(String dst) {
      label = dst;
    }
    String readLabel() { return label; }
  }
public Destination to(String s) {
  return new Destination(s);
}
public Contents cont() {
  return new Contents();
}
public void ship(String dest) {
  Contents c = cont();
  Destination d = to(dest);
  System.out.println(d.readLabel());
}
public static void main(String[] args) {
  Parcel p = new Parcel();
  p.ship("Romania");
  Parcel q = new Parcel();
  // Defining references to inner classes:
  Parcel.Contents c = q.cont();
  Parcel.Destination d = q.to("China");
}
}
```

Inner Classes

- Inner classes come in four flavors:
  - Static member classes
  - Member classes
  - Local classes
  - Anonymous classes
- A *static member class* is a static member of a class.
  - Has access to all *static* methods of the outer, or top-level, class

```
class MyOuter {
    public static class MyInner {
        //...
        public void function1() {}
        //more functions
    }
}
```

Inner Classes

- A *member class* is also defined as a member of a class.
  - Unlike the static variety, the member class is *instance specific* and
  - has *access* to any and *all* methods and members, even the outer's *this* reference.

```
class MyOuter {
   private float variable = 0;
   public void doSomething() { //do your stuff }
   private class MyInner  {
      public void doSomething() {//do your stuff }
      public void function() {
         //To call a function with the same name from the enclosing class
        MyOutter.this.doSomething();
      }
   }
}
```

Inner Classes

- *Local classes* are declared within a block of code and are visible only within that block, just as any other method variable.

```
interface MyInterface {
   public String getInfo();
}
class MyOuter {
   MyInterface current_object;
   public void setInterface(String info)   {
      class MyInner implements MyInterface   {
         private String info;
         public MyInner(String inf) {info=inf;}
         public String getInfo() {return info;}
      }
      current_object = new MyInner(info);
   }
}
```

Inner Classes

- An *anonymous* class is a local class that has no name
- Inner class are very useful when you want to implement callbacks.
  - If you try to implement a callback procedure without an inner class, then when you implement the **`ActionListener`** interface in a class this would force you to use a lot of **`if`** and **`else if`** to figure out on which object the event occurred.
  - Using inner classes allows you to efficiently have a separate block of code to handle the **`actionPerformed`** function for every graphic component object.

Array Basics

- In Java, an *array* is an indexed collection of data values of the same type.
- Arrays are useful for storing and manipulating a collection of values.
- In Java, an array is a reference data type.
- We use the **new** operator to allocate the memory to store the values in an array.

```
rainfall = new double [12];
 //creates an array of size 12.
```

- We use an *indexed expression* to refer to the individual values of the collection. Arrays use zero-based indexing.

Array Basics

- An array has a *public constant* **length** for the size of an array.
- Do not confuse the **length** *value* of an array and the **length** *method* of a **String** object.
- The length is a method for a **String** object, so we use the syntax for calling a method.

```
String str = "This is a string";
int size = str.length();
```

- An array, on the other hand, is a reference data type, not an object. Therefore, we do not use a method call.

```
int size = rainfall.length;
```

Array Basics

- Using constants to declare the array sizes does not always lead to efficient use of space.
- Declaration of arrays with constants is called *fixed-size array declaration*.
- Fixed-size array declaration may pose two problems:
  - Not enough capacity for the task at hand.
  - Wasted space.
- In Java, we are not limited to fixed-size array declaration.

Array Basics

- The following code prompts the user for the size of an array and declares an array of designated size:

```
int size;
int[] number;
size= Integer.parseInt(JOptionPane.showInputDialog(null,
            "Size of an array:"));
number = new int[size];
```

- Any valid integer arithmetic expression is allowed for specifying the size of an array:

```
size = Integer.parseInt(
      JOptionPane.showInputDialog(null,""));
number = new int[size*size + 2* size + 5];
```

- Arrays are not limited to primitive data types.

Arrays are Objects

`int[] data;`	*data is a reference variable whose type is int[], meaning "array of ints". At this point its value is null.*
`data = new int[5];`	*The new operator causes a chunk of memory big enough for 5 ints to be allocated on the heap. Here, data is a assigned a reference to the heap address.*
`data[0] = 6;` `data[2] = 12;`	*Initially, all five ints are 0. Here, two of them are assigned other values.*
`int[] info = {6, 10, 12, 0, 0};`	
`int[] info = new int[]{6, 10, 12, 0, 0};`	

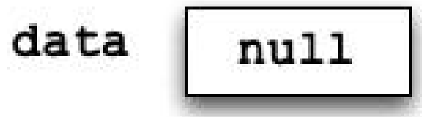
Array Out of Bounds Exceptions

```
public class ArrayTool{
  public int sum(int[] data){
    int sum = 0;
    for (int i = 0; i < data.length; i++){
      sum += data[i];
    }
    return sum;
  }
  public int sum2(int[] data){
    int sum = 0;
    for (int i = 0; i <= data.length; i++){
      sum += data[i];
    }
    return sum;
  }
}
```

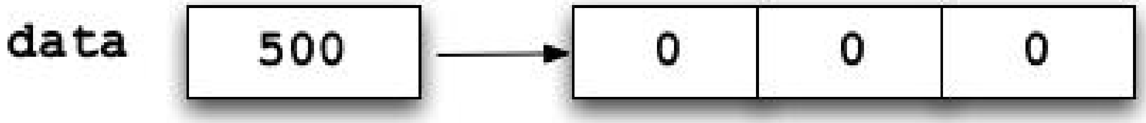
Use of this comparison causes an ArrayIndexOutOfBoundsException to be thrown

Array of Primitives

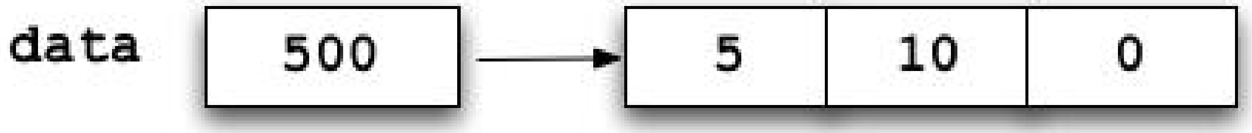
```
int[] data;
```



```
data = new int[3];
```



```
data[0] = 5;
data[1] = 10;
```



Arrays of Other Primitive Types

```
double[] temps;
temps = new double[24];
temps[0] = 78.5;
temps[1] = 84.2;
```

```
boolean[] answers = new boolean[6];
. . .
if (answers[0])
   doSmth();
```

```
char[] buffer = new char[500];
open a file for reading
while (more chars in file & buffer not full)
    buffer[i++] = char from file
```

2D Arrays

- Array can have 2, 3, or more dimensions
- When declaring a variable for such an array, use a pair of square brackets for each dimension
- For 2D arrays, the elements are indexed [row][column]
- Example:

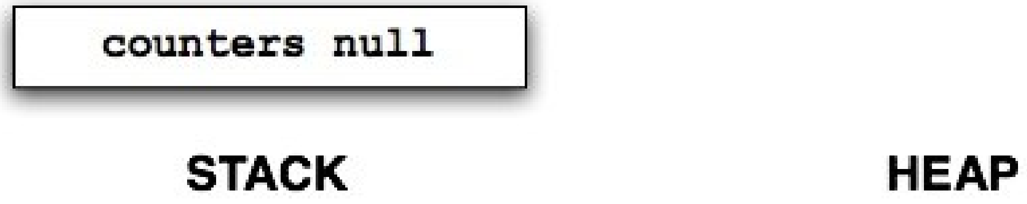
```
char[][] board;
board = new char[3][3];
board[1][1] = 'X';
board[0][0] = 'O';
board[0][1] = 'X';
```

A Counter Class

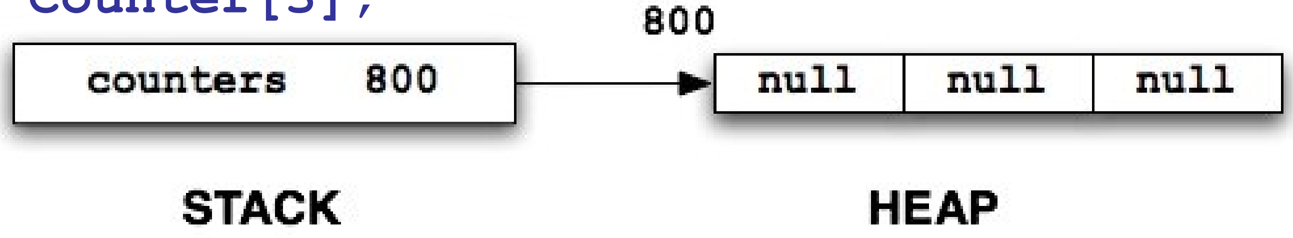
```
public class Counter
{
private int count;
/**
*  Constructor. Initializes
   count to zero.
*/
public Counter()
{
    count = 0;
}
 /**
* @return The current count for
   this type.
*/
public int getCount()
{
     return count;
}
/**
* Increment the current count by
  one.
*/
public void increment()
{
    count++;
}
/**
* Reset the current count to zero.
*/
 public void reset()
{
  count = 0;
}
}
```

Array of Objects

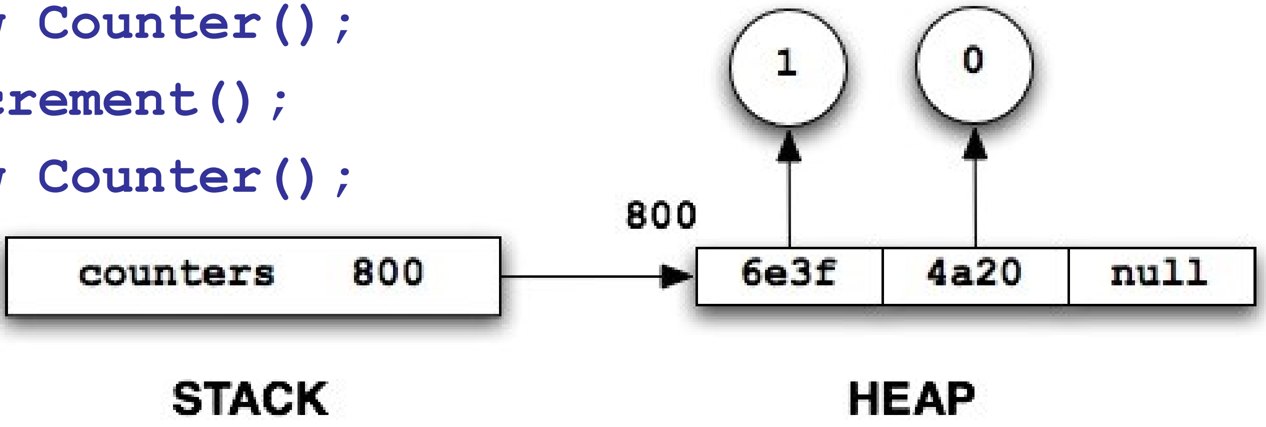
```
Counter[] counters;
```



```
counters = new Counter[3];
```



```
counters[0]=new Counter();
counters[0].increment();
counters[1]=new Counter();
```



"Traversing" Arrays of Objects

- As we can use loops to traverse arrays of primitives, we can do the same with arrays of objects
- Exercise. Write a method that
  - takes one argument: an array of counters
  - returns the sum of numbers contained in the counters
  - first, assume that each array element points to a valid counter
  - Then rewrite the method so it will handle arrays for which some or all of its elements are null

Pitfall: An Array of Characters Is Not a String

- An array of characters is conceptually a list of characters, and so is conceptually like a string
- However, an array of characters is not an object of the class `String`

```
char[] a = {'A', 'B', 'C'};
String s = a; //Illegal!
```

- An array of characters can be converted to an object of type `String`, however

Pitfall: An Array of Characters Is Not a String

- The class `String` has a constructor that has a single parameter of type `char[]`

```
String s = new String(a);
```

  - The object `s` will have the same sequence of characters as the entire array `a` (`"ABC"`), but is an *independent* copy

- Another `String` constructor uses a subrange of a character array instead

```
String s2 = new String(a,0,2);
```

  - Given `a` as before, the new string object is `"AB"`

Pitfall: An Array of Characters Is Not a String

- An array of characters does have some things in common with `String` objects
  - For example, an array of characters can be output using `println`

```
System.out.println(a);
```

  - Given `a` as before, this would produce the output

```
ABC
```

Shortcuts for Initializing Arrays

Arrays of Primitives:

```
int[] info1 =               { 2000, 100, 40, 60};
int[] info2 = new int[]{ 2000, 100, 40, 60};
char[] choices1 =               { 'p', 's', 'q'};
char[] choices2 = new char[]{ 'p', 's', 'q'};
double[] temps1 =               {75.6, 99.4, 86.7};
double[] temps2 = new double[] {75.6, 99.4, 86.7};
```

Shortcuts for Initializing Arrays

Arrays of Objects:

```
Person[] people = {new Person("jo"),new Person("flo")};
Person[] people = new Person[] {new Person("jo"),
                                new Person("flo")};
Point p1 = new Point(0,0);
Point[] points1 = {p1, new Point(0, 10)};
Point[] points2 = new Point[] {p1, new Point(0, 10)};
```

Note: The use of "new type[ ]" syntax is that it can be used in an assignment statement that is not a variable declaration statement.

Passing Arrays as Parameters

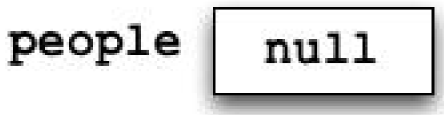
- When an object has no references pointing to it, the system will erase the object and make the memory space available again.
- Erasing an object is called *deallocation* of memory.
- The process of deallocating memory is called *garbage collection*. Garbage collection is automatically performed in Java.
- When an array is passed to a method, only its reference is passed.
- A copy of the array is not created in the method.

Example: Person Database

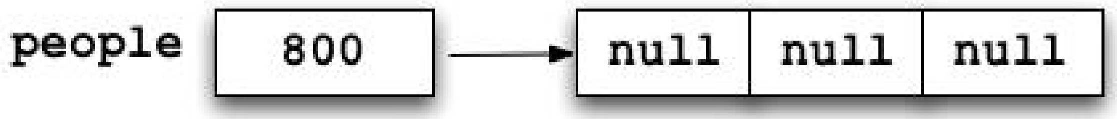
```
public class Person{
  private String name;
  private int age;
  public Person(String name, int age){
    this.name = name;
    this.age = age;
  }
  public Person(String name){
    this(name, 5);
  }
  public int getAge() { return age; }
  public String getName() { return name; }
}
```

Array of Objects

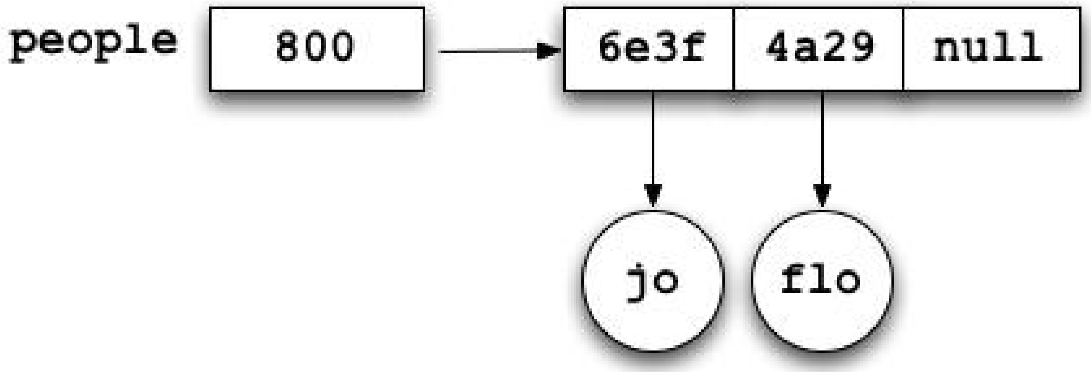
```
Person[] people;
```



```
people = new Person[3];
```



```
people[0]=new Person(“jo”);
people[1]=new
Person(“flo”);
```



Example: Person Database

```
public class PersonDB{
  private Person[] people;
  public PersonDB(){
    people = new Person[]{new Person("jo",25),
                new Person("flo",18),
                new Person("mo", 19)};
  }
   /** Calculates and returns the average age. */
  public double getAverageAge(){
    return 0; // DIY
  }
/** Returns true if name is in database, otherwise false */
  public boolean isInDatabase(String searchName){
    return false; // DIY
  }
}
```

Copying arrays

- `System` class has an `arraycopy` method
  - Used to efficiently copy data from one array to another

```
public static void arraycopy(Object src,
   int srcPos, Object dest, int destPos, int
   length)
```

Reading

- Deitel: chapters 6, 7
- Barnes: 4.16, 6
- Eckel: chapters 6, 7, 11, 17

Summary

- Methods: How call works
- Creating objects
- Class design hints
- Kinds of classes
  - Inner classes
- Arrays:
  - Basics
  - Arrays of primitives
  - Arrays of objects
  - Traversing
  - Initializing
  - Passing arrays a parameters