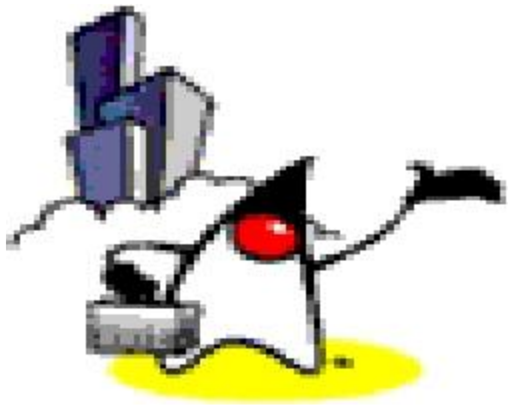
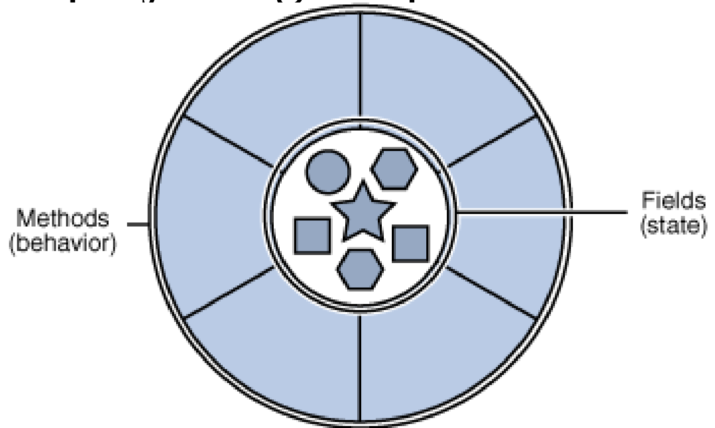
Working with Java Classes



Brief Introduction on OOP

Introduction to OOP

- Object-Oriented programming or OOP
  - Revolves around the concept of objects as the basic elements of your programs.
  - These objects are characterized by their properties and behaviors.
  - An object is a software bundle of related state and behavior. Software objects are often used to model the real-world objects that you find in everyday life.
  - **Abstraction:** The process of hiding unwanted/implementation details and projecting the public interface to communicate with.



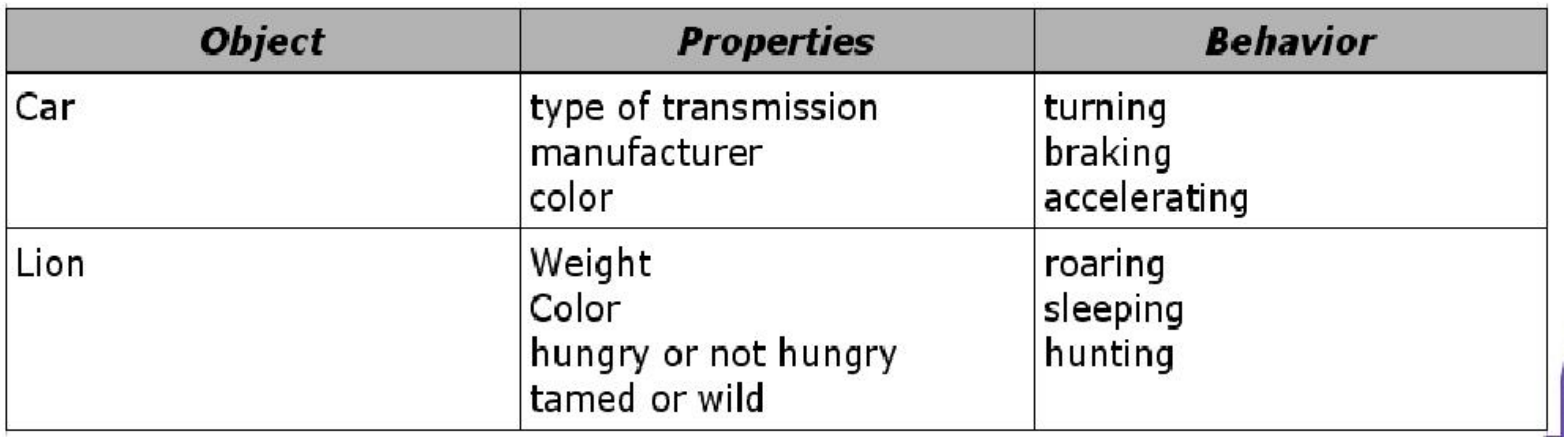
- Encapsulation
  - Restricting access to some of the object's components.
  - Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition. Typically, only the object's own methods can directly inspect or manipulate its fields.
- Polymorphism

  Greek meaning “many forms In Java, polymorphism
  refers to the ‘dynamic binding’ mechanism that determines which
  method definition will be used when a method name has been overridden.
  - Can treat an object of a subclass as an object of its superclass.
  - A reference variable of a superclass type can point to an object
    of its subclass.

- Inheritance
  - Object-oriented programming allows classes/objects to *inherit* commonly used state and behavior from other classes. Inheritance provides a powerful and natural mechanism for organizing and structuring your software.
    - - Parent class is called SuperClass or base class.
    - - Child class is called Sub-class or derived class.
    - - IS-A relationship.
- Overriding
  - Method overriding, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by its superclass or parent class. The implementation in the subclass overrides (replaces) the implementation in the superclass by providing a method that has same name, same parameters or signature, and same return type as the method in the parent class.

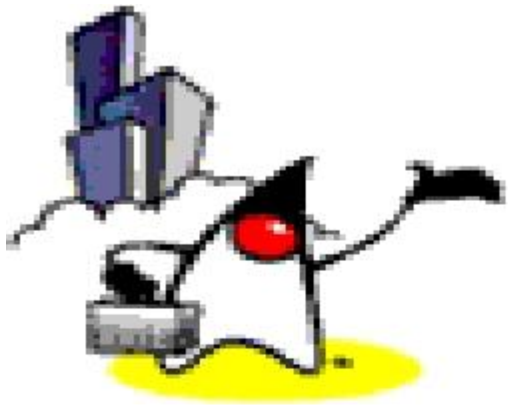
Introduction to OOP

- Example of objects



Object	*Properties*	*Behavior*
Car	type of transmission manufacturer color	turning braking accelerating
Lion	Weight Color hungry or not hungry tamed or wild	roaring sleeping hunting

objects in the physical world can easily be modeled as software objects using the properties as data and the behaviors as methods



Classes and Objects (Object Instances)

Classes and Objects

- Class
  - can be thought of as a template, a prototype or a blueprint of an object
  - is the fundamental structure in object-oriented programming

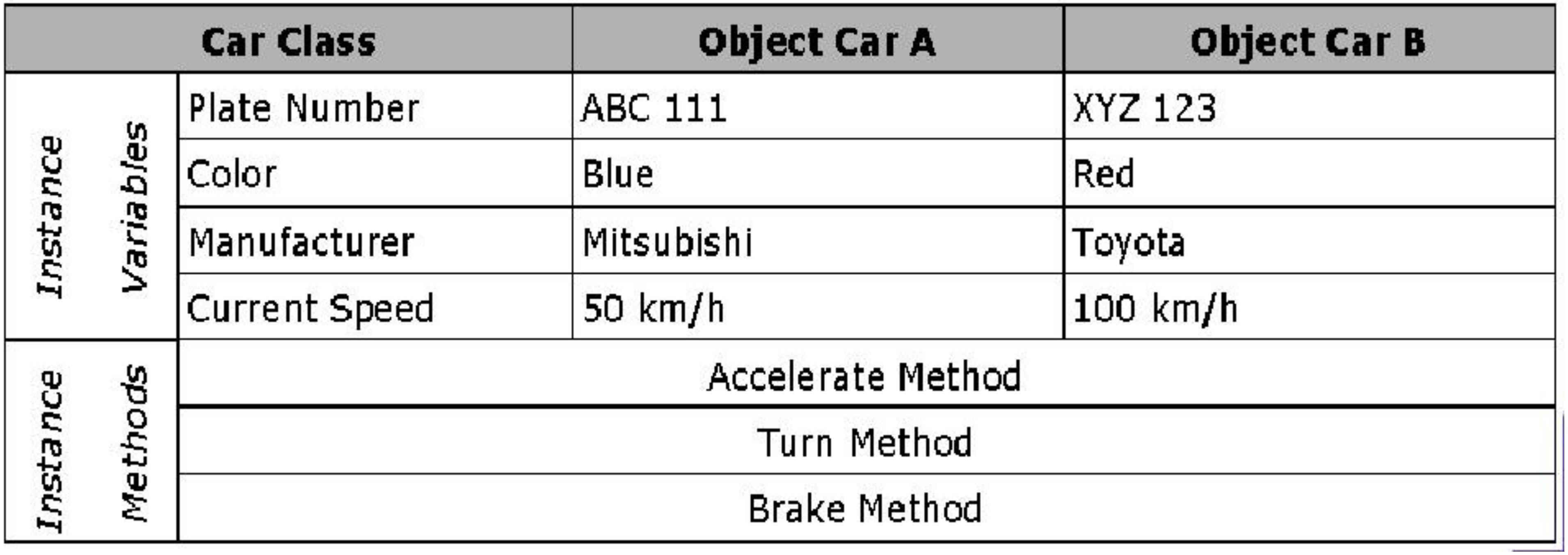
- Two types of class members:
  - Fields (properties, variables)
    - specify the data types defined by the class
  - Methods (behavior)
    - specify the operations

Classes and Objects

- Object
  - An object is an instance of a class - we will call it object instance
  - The property values of an object instance is different from the ones of other object instances of a same class
  - Object instances of a same class share the same behavior (methods)

Classes and Objects

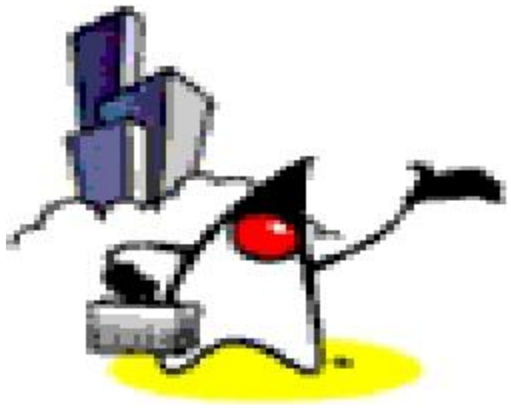
- To differentiate between classes and objects, let us discuss an example:



Car Class		Object Car A	Object Car B
Instance Variables	Plate Number	ABC 111	XYZ 123
	Color	Blue	Red
	Manufacturer	Mitsubishi	Toyota
	Current Speed	50 km/h	100 km/h
Instance Methods	Accelerate Method		
	Turn Method		
	Brake Method		

Classes and Objects

- Classes provide the benefit of reusability.
- Software programmers can use a class over and over again to create many object instances.



Defining Your Own Class

Defining your own classes

- Things to take note of for the syntax defined in this section:

*	means that there may be 0 or more occurrences of the line where it was applied to.
<description>	indicates that you have to substitute an actual value for this part instead of typing it as it is.
[]	indicates that this part is optional

Defining your own classes

- To define a class, we write:

```
<modifier> class <name> {
      <attributeDeclaration>*
      <constructorDeclaration>*
      <methodDeclaration>*
}
```

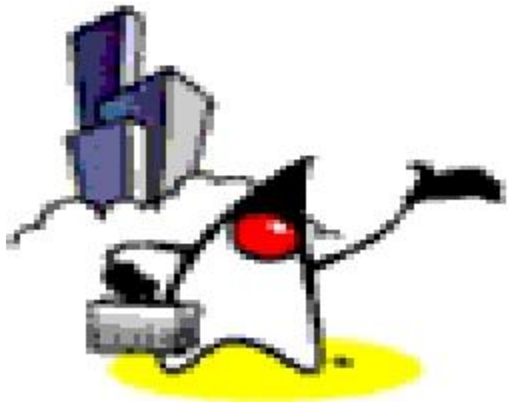
- where
  - <modifier> is an access modifier, which may be combined with other types of modifier.

Example

```
public class StudentRecord {
    //we'll add more code here later

}
```

□ where,

- public - means that our class is accessible to other classes outside the package
- class - this is the keyword used to create a class in Java
- StudentRecord - a unique identifier that describes our class



Creation of Object Instances with "new" keyword

Creation of Object Instance

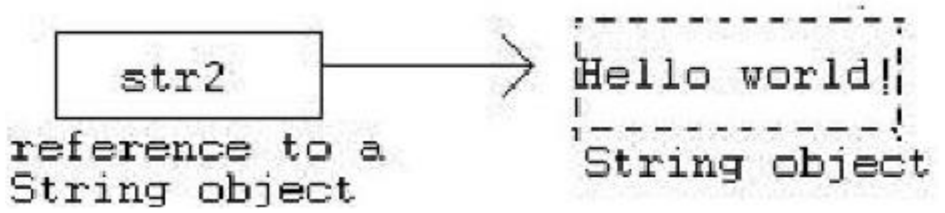
- To create an object instance of a class, we use the **new** operator.
- For example, if you want to create an instance of the class String, we write the following code,

```
String str2 = new String("Hello world!");
```

  or also equivalent to,

```
String str2 = "Hello";
```

- String class is a special (and only) class you can create an instance without using **new** keyword as shown above



Creation of Object Instance

- The **new** operator
  - allocates a memory for that object and returns a reference of that memory location to you.
  - When you create an object, you actually invoke the class' constructor.

- The constructor
  - is a method where you place all the initializations, it has the same name as the class.

Constructors (Constructor Methods)

Constructors

- Constructors are important in instantiating an object. It is a method where all the initializations are placed.

- The following are the properties of a constructor:
  - Constructors have the same name as the class
  - A constructor is just like an ordinary method, however only the following information can be placed in the header of the constructor,
  - scope or accessibility identifier (like public...), constructor's name and parameters if it has any.
  - Constructors does not have any return value
  - You cannot call a constructor directly, it can only be called by using the new operator during class instantiation.

Constructors

- To declare a constructor, we write,

```
<modifier> <className>  (<parameter>*) {
       <statement>*
}
```

Default Constructor (Method)

- The **default constructor (no-arg constructor)**
  - is the constructor without any parameters.
  - If the class does not specify any constructors, then an implicit default constructor is created.

Example: Default Constructor Method of StudentRecord Class

```
public StudentRecord()
{
    //some code here
}
```

Overloading Constructor Methods

```
public StudentRecord(){
     //some initialization code here
}
public StudentRecord(String temp){
     this.name = temp;
}
public StudentRecord(String name, String address){
     this.name = name;
     this.address = address;
}
public StudentRecord(double mGrade, double eGrade,
                   double sGrade){
     mathGrade = mGrade;
     englishGrade = eGrade;
     scienceGrade = sGrade;

}
```

Using Constructors

- To use these constructors, we have the following code,

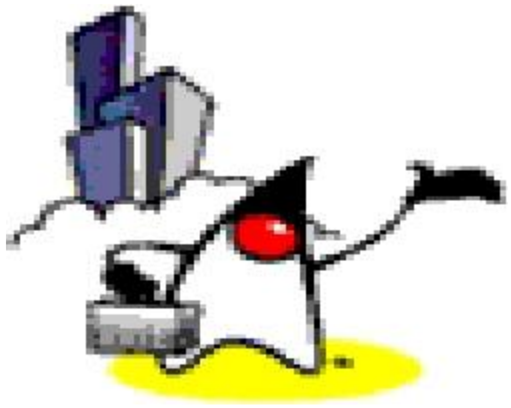
```
public static void main( String[] args ){
    //create three objects for Student record
    StudentRecord annaRecord=new StudentRecord("Anna");
    StudentRecord beahRecord=new StudentRecord("Beah",
                         "Philippines");
    StudentRecord crisRecord=new
      StudentRecord(80,90,100);
    //some code here
}
```

“this()” constructor call

- Constructor calls can be chained, meaning, you can call another constructor from inside a constructor.
- We use the this() call for this
- There are a few things to remember when using the this() constructor call:
  - When using the this constructor call, IT MUST OCCUR AS THE FIRST STATEMENT in a constructor
  - It can ONLY BE USED IN A CONSTRUCTOR DEFINITION. The this call can then be followed by any other relevant statements**.**

Example

```
public StudentRecord(){
    this("some string");
}

public StudentRecord(String temp){
     this.name = temp;
}

public static void main( String[] args )
{

    StudentRecord    annaRecord = new StudentRecord();
}
```



“this” Reference

“this” reference

- The this reference
  - refers to current object instance itself
  - used to access the instance variables shadowed by the parameters.

- To use the this reference, we type,

```
this.<nameOfTheInstanceVariable>
```

- You can only use the this reference for instance variables and NOT static or class variables.

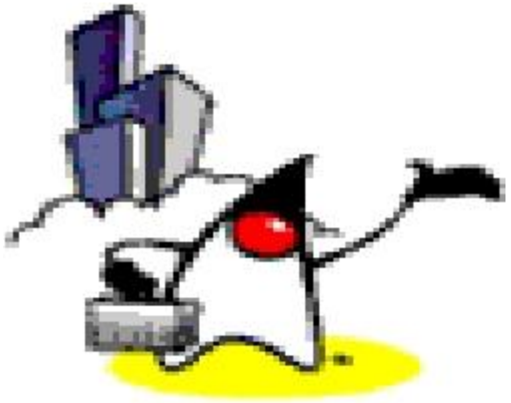
“this” reference

- The this reference is assumed when you call a method from the same object

```
public class MyClass {
    void aMethod() {
        // same thing as this.anotherMethod()
        anotherMethod();
    }
    void anotherMethod() {
      // method definition here...
    }
}
```

Example

```
public void setAge( int age ){
    this.age = age;
}
```



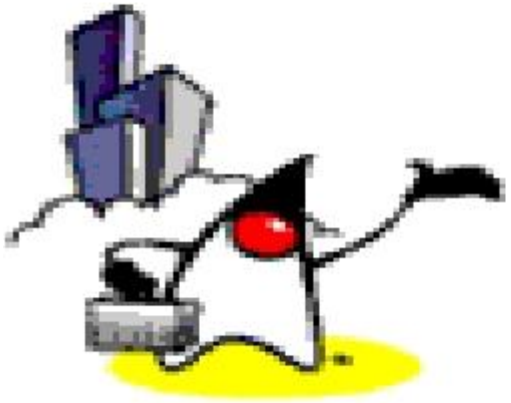
Java Variables

Variables

- Data identifiers
- Are used to refer to specific values that are generated in a program--values that you want to keep around

Three Types of Variables

- Local (automatic) variable
  - declared within a method body
  - visible only within a method body
  - maintained in a stack
- Instance variable
  - declared inside a class body but outside of any method bodies
  - per each object instance
  - cannot be referenced from static context
- Class (static) variable
  - declared inside a class body but outside of any method bodies
  - prepended with the static modifier
  - per each class
  - shared by all object instances



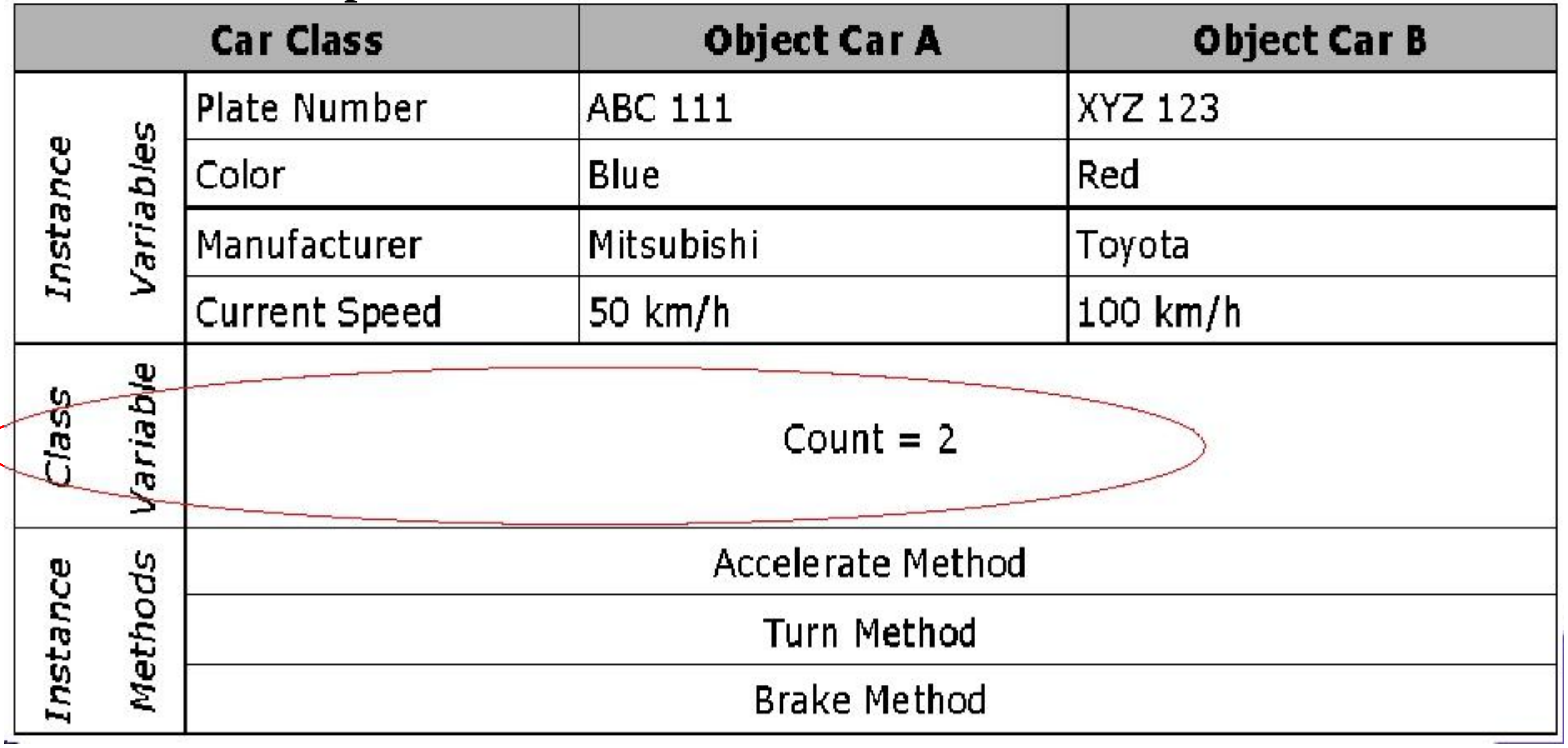
Instance Variables vs. Static Variables

Instance Variables (Properties) vs. Class (Static) Variables

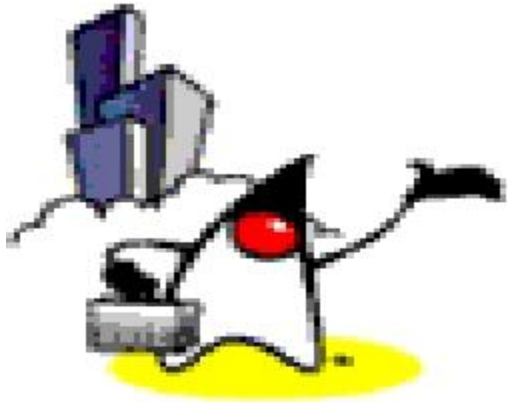
- Instance Variables
  - Belongs to an object instance
  - Value of variable of an object instance is different from the ones of other object object instances
- Class Variables (also called static member variables)
  - variables that belong to the whole class.
  - This means that they have the same value for all the object instances in the same class.

Class Variables

- For example,



Car Class		Object Car A	Object Car B
Instance Variables	Plate Number	ABC 111	XYZ 123
	Color	Blue	Red
	Manufacturer	Mitsubishi	Toyota
	Current Speed	50 km/h	100 km/h
Class Variable	Count = 2		
Instance Methods	Accelerate Method		
	Turn Method		
	Brake Method		



Instance Variables

Declaring Properties (Attributes)

- To declare a certain attribute for our class, we write,

```
<modifier> <type> <name> [=
  <default_value>];
```

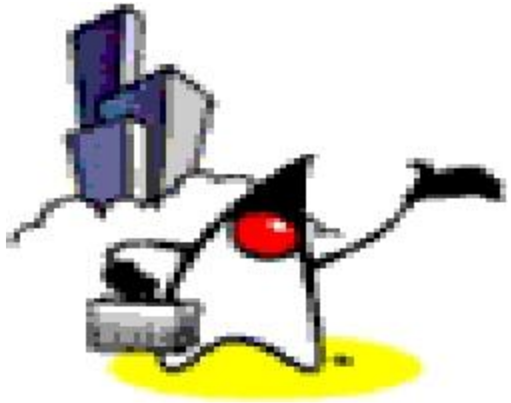
Instance Variables

```
public class StudentRecord {
    // Instance variables
     private String    name;
     private String    address;
     private int    age;
     private double mathGrade;
     private double englishGrade;
     private double scienceGrade;
     private double average;
     //we'll add more code here later
}
```

- where,
  - private here means that the variables are only accessible within the class. Other objects cannot access these variables directly. We will cover more about accessibility later.

Coding Guidelines

- Declare all your instance variables right after “public class Myclass {“
- Declare one variable for each line.
- Instance variables, like any other variables should start with a SMALL letter.
- Use an appropriate data type for each variable you declare.
- Declare instance variables as private so that only class methods can access them directly.
    - Encapsulation

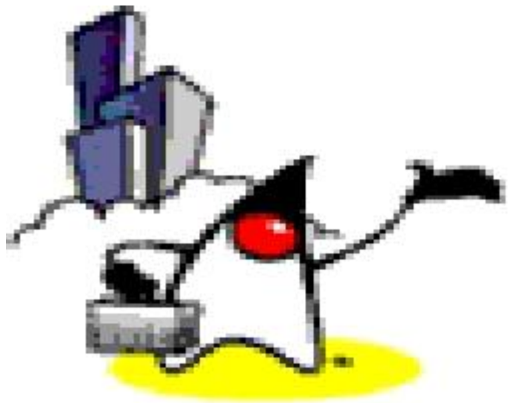


Static Variables

Class (static) variables

```
public class StudentRecord {
    //static variables we have declared
    private static int studentCount;
    //we'll add more code here later
}
```

- we use the keyword static to indicate that a variable is a static variable.



Methods (Instance methods & Static methods)

Methods

- Method
  - is a separate piece of code that can be called by a main program or any other method to perform some specific function.

- The following are characteristics of methods:
  - It can return one or no values
  - It may accept as many parameters it needs or no parameter at all. Parameters are also called arguments.
  - After the method has finished execution, it goes back to the method that called it.

Why Use Methods?

- Methods contain behavior of a class (business logic)
  - The heart of effective problem solving is in problem decomposition.
  - We can do this in Java by creating methods to solve a specific part of the problem.
  - Taking a problem and breaking it into small, manageable pieces is critical to writing large programs.

Declaring Methods

- To declare methods we write,

```
<modifier> <returnType>
  <name>(<parameter>*) {
        <statement>*
}
```

- where,
  - <modifier> can carry a number of different modifiers
  - <returnType> can be any data type (including void)
  - <name> can be any valid identifier
  - <parameter> ::= <parameter_type> <parameter_name>[,]

Coding Guidelines

- Method names should start with a SMALL letter.
- Method names should be verbs
- Always provide documentation before the declaration of the method. You can use Javadocs style for this. Please see example.

Accessor (Getter) Methods

- Accessor methods
  - used to read values from our class variables (instance/static).
  - usually written as:

    `get<NameOfInstanceVariable>`
  - It also returns a value.

Example 1: Accessor (Getter) Method

```
public class StudentRecord {
    private String  name;
    :
    public String getName(){
        return name;
    }
}
```

- where,
  - public - means that the method can be called from objects outside the class
  - String - is the return type of the method. This means that the method should return a value of type String
  - getName - the name of the method
  - () - this means that our method does not have any parameters

Example 2: Accessor (Getter) Method

```
public class StudentRecord {
    private String  name;
    // some code

    // An example in which the business logic is
    // used to return a value on an accessor method
    public double getAverage(){
        double result = 0;
        result=(mathGrade+englishGrade+scienceGrade)/3;
        return result;
    }
}
```

Mutator (Setter) Methods

- Mutator Methods
  - used to write or change values of our class variables (instance/static).
  - Usually written as:

```
set<NameOfInstanceVariable>
```

Example: Mutator (Setter) Method

```
public class StudentRecord {
    private String  name;
    :
    public void setName( String temp ){
        name = temp;
    }
}
```

- where,
  - public - means that the method can be called from objects outside the class
  - void - means that the method does not return any value
  - setName - the name of the method
  - (String temp) - parameter that will be used inside our method

Multiple return statements

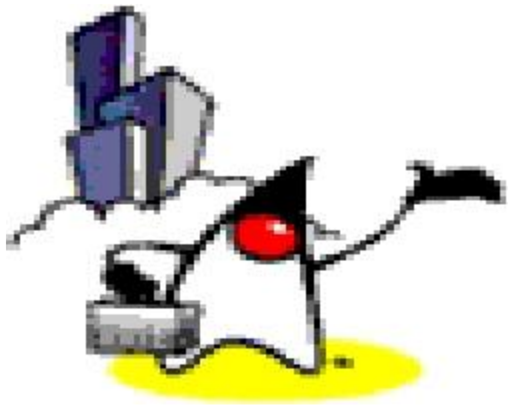
- You can have multiple return statements for a method as long as they are not on the same block.
- You can also use constants to return values instead of variables.

Example: Multiple return statements

```
public String getNumberInWords( int num ){
    String defaultNum = "zero";
    if( num == 1 ){
        return "one"; //return a constant
    }
    else if( num == 2){
        return "two"; //return a constant
    }
    //return a variable
    return defaultNum;
}
```

Two Types of Methods

- Instance (non-static) methods
  - Should be called after object instance is created
  - More common than static methods
- Static methods
  - Should be called in the form of [ClassName].[methodName]



Static Methods

Static methods

```
public class StudentRecord {
    private static int studentCount;
    public static int getStudentCount(){
        return studentCount;
    }
}
```

- where,
  - public- means that the method can be called from objects outside the class
  - static-means that the method is static and should be called by typing,[ClassName].[methodName]. For example, in this case, we call the method `StudentRecord.getStudentCount()`
  - int- is the return type of the method. This means that the method should return a value of type int
  - getStudentCount- the name of the method
  - ()- this means that our method does not have any parameters

When to Define Static Method?

- When the logic and state does not involve specific object instance
  - Computation method
  - add(int x, int y) method
- When the logic is a convenience without creating an object instance
  - Integer.parseInt();

Source Code for StudentRecord class

```
public class StudentRecord {

    // Instance variables
     private String    name;
     private String    address;
     private int    age;
     private double mathGrade;
     private double englishGrade;
     private double scienceGrade;
     private double average;
     private static int studentCount;
```

Source Code for StudentRecord Class

```
/**
 * Returns the name of the student (Accessor method)
 */
public String getName(){
    return name;
}

/**
 * Changes the name of the student (Mutator method)
 */
public void setName( String temp ){
    name = temp;
}
```

Source Code for StudentRecord Class

```
/**
 * Computes the average of the english, math and science
 * grades (Accessor method)
 */
public double getAverage(){
    double result = 0;
    result = ( mathGrade+englishGrade+scienceGrade )/3;
    return result;
}
/**
 * returns the number of instances of StudentRecords
 * (Accessor method)
 */
public static int getStudentCount(){
    return studentCount;
}
```

Calling Instance (non-static) Methods

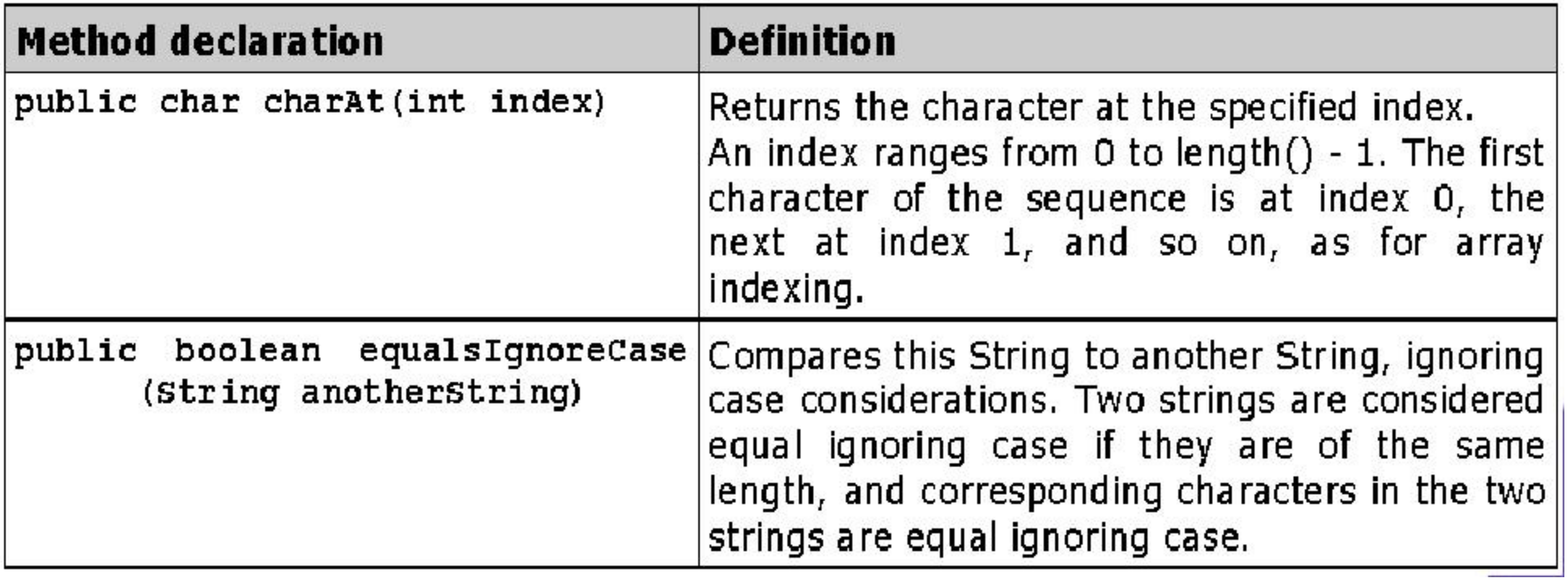
- To illustrate how to call methods, let's use the **String** class as an example.
- You can use the Java API documentation to see all the available methods in the String class.
- Later on, we will create our own methods, but for now, let us use what is available.
- To call an instance method, we write the following,

  nameOfObject.nameOfMethod(parameters);
- Example

```
String strInstance1 = new String("I am object
   instance of a String class");

char x = strInstance1.charAt(2);
```

Calling Instance Methods

- Let's take two sample methods found in the String class



Method declaration	Definition
`public char charAt(int index)`	Returns the character at the specified index. An index ranges from 0 to length() - 1. The first character of the sequence is at index 0, the next at index 1, and so on, as for array indexing.
`public boolean equalsIgnoreCase (String anotherString)`	Compares this String to another String, ignoring case considerations. Two strings are considered equal ignoring case if they are of the same length, and corresponding characters in the two strings are equal ignoring case.

Calling Instance Methods

- Example

```
String  str1  =  "Hello";

char     x  =  str1.charAt(0);  //will  return  the  character  H
                                        //and  store  it  to  variable  x

String       str2  =  "hello";

//this  will  return  a  boolean  value  true
boolean      result  =  str1.equalsIgnoreCase(  str2  );
```

Calling Static Methods

- Static methods
  - methods that can be invoked without instantiating a class (means without invoking the new keyword).
  - Static methods belong to the class as a whole and not to a certain instance (or object) of a class.
  - Static methods are distinguished from instance methods in a class definition by the keyword static.

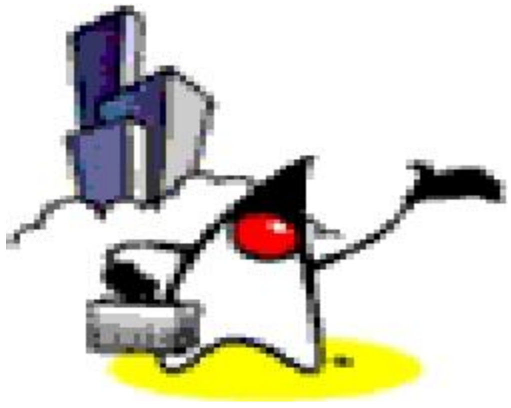
- To call a static method, just type,

```
Classname.staticMethodName(params);
```

Calling Static Methods

- Examples of static methods, we've used so far in our examples are,

```
//prints data to screen
System.out.println("Hello world");

//converts the String 10, to an integer
int i = Integer.parseInt("10");

//Returns a String representation of the integer argument as an
//unsigned integer base 16
String hexEquivalent = Integer.toHexString( 10 );
```

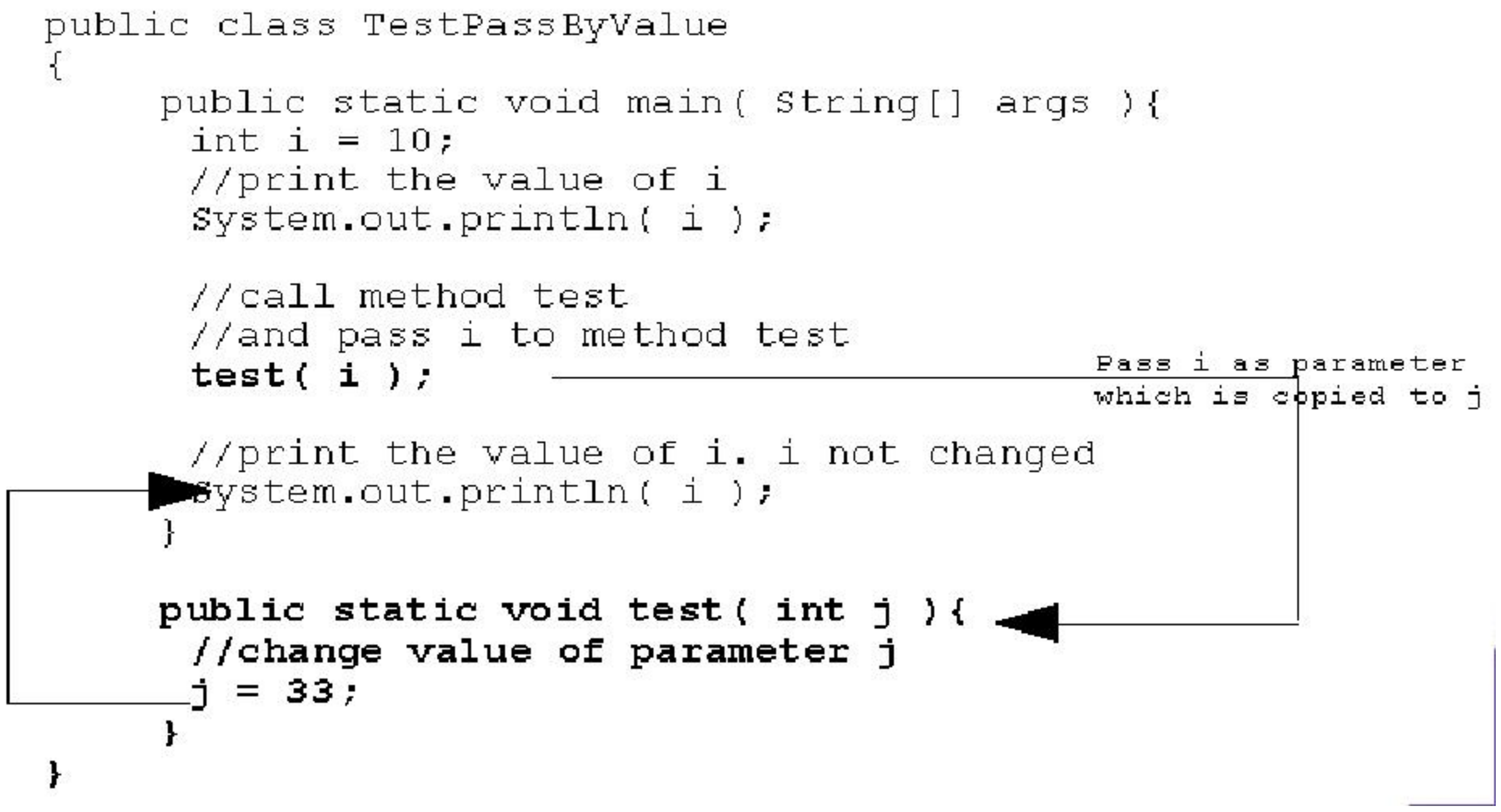


Parameter Passing (Pass-by-value & Pass-by-reference)

Parameter Passing

- Pass-by-Value
  - when a pass-by-value occurs, the method makes a copy of the value of the variable passed to the method. The method cannot accidentally modify the original argument even if it modifies the parameters during calculations.
  - all primitive data types when passed to a method are pass-by-value.

Pass-by-Value



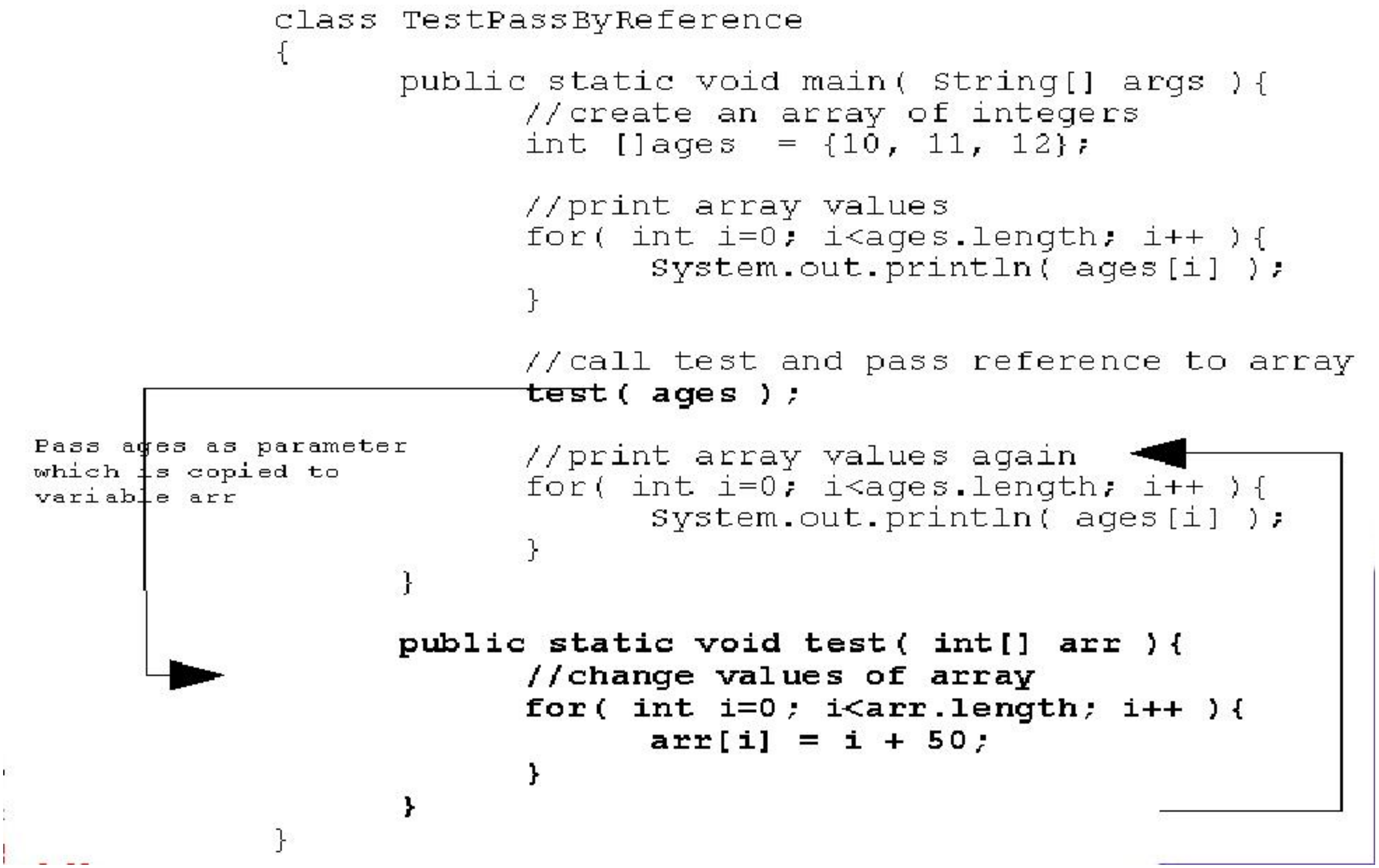
```
public class TestPassByValue
{
      public static void main( String[] args ){
       int i = 10;
       //print the value of i
       System.out.println( i );

       //call method test
       //and pass i to method test
       test( i );

       //print the value of i. i not changed
       System.out.println( i );
      }

      public static void test( int j ){
       //change value of parameter j
       j = 33;
      }
}
```

Pass i as parameter
which is copied to j

Parameter Passing

- Pass-by-Reference
  - When a pass-by-reference occurs, the reference to an object is passed to the calling method. This means that, the method makes a copy of the reference of the variable passed to the method.
  - However, unlike in pass-by-value, the method can modify the actual object that the reference is pointing to, since, although different references are used in the methods, the location of the data they are pointing to is the same.

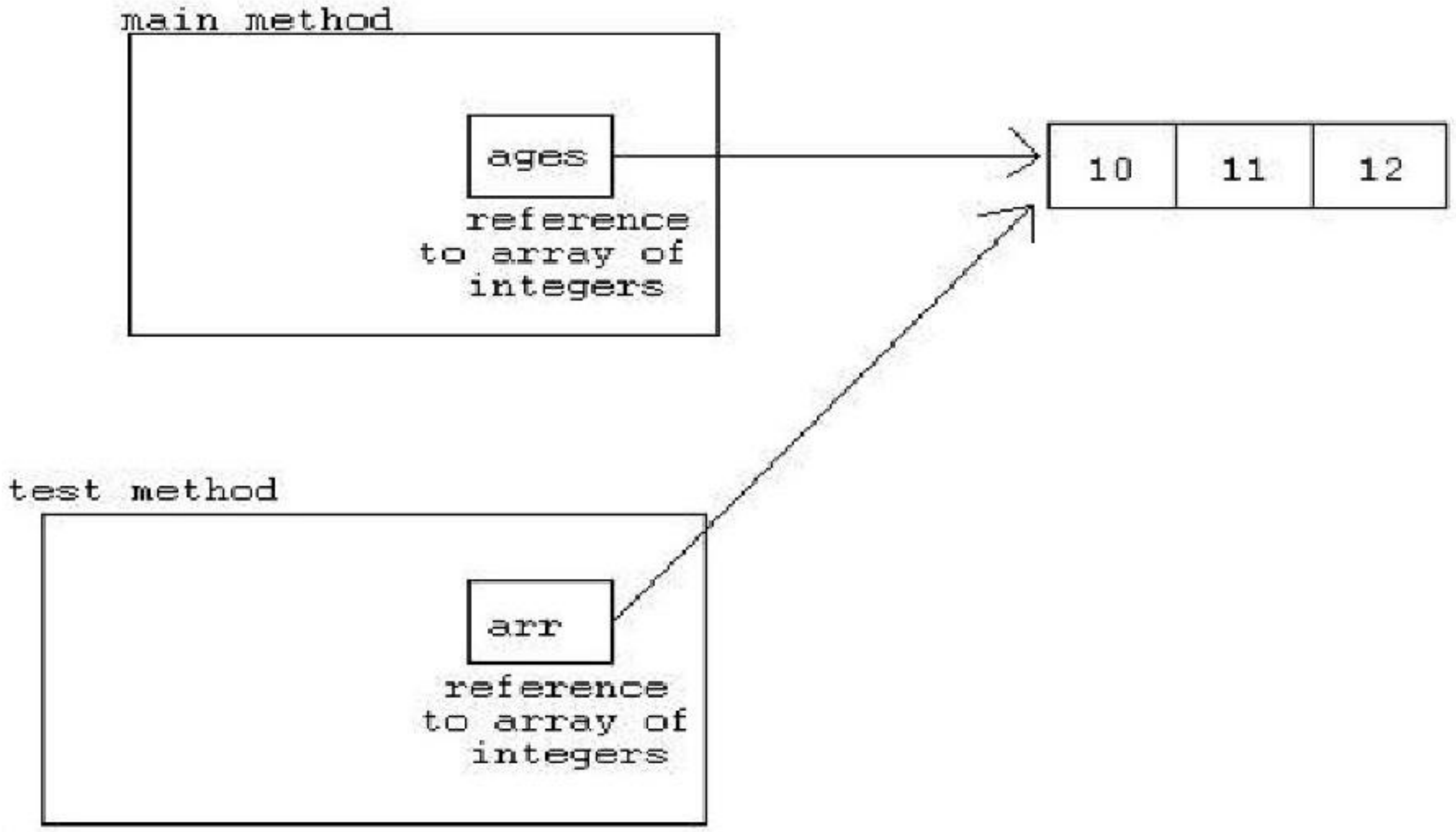
Pass-by-Reference

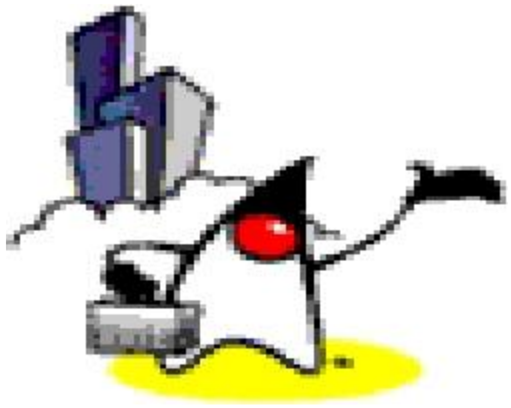


```
class TestPassByReference
{
        public static void main( String[] args ){
                //create an array of integers
                int []ages  = {10, 11, 12};

                //print array values
                for( int i=0; i<ages.length; i++ ){
                        System.out.println( ages[i] );
                }

                //call test and pass reference to array
                test( ages );

                //print array values again
                for( int i=0; i<ages.length; i++ ){
                        System.out.println( ages[i] );
                }
        }

        public static void test( int[] arr ){
                //change values of array
                for( int i=0; i<arr.length; i++ ){
                        arr[i] = i + 50;
                }
        }
}
```

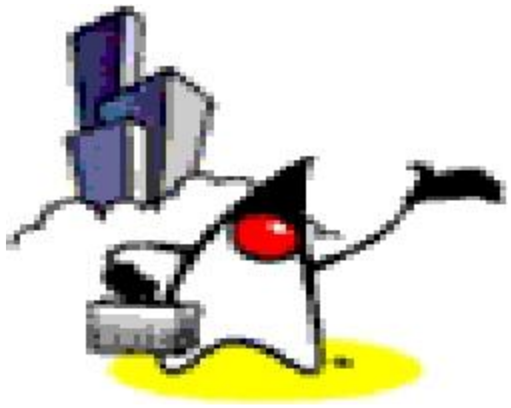
Pass ages as parameter which is copied to variable arr

Pass-by-Reference


main method
ages
reference to array of integers
10
11
12
test method
arr
reference to array of integers



Variables (Fields, Properties)



Scope of a Variable

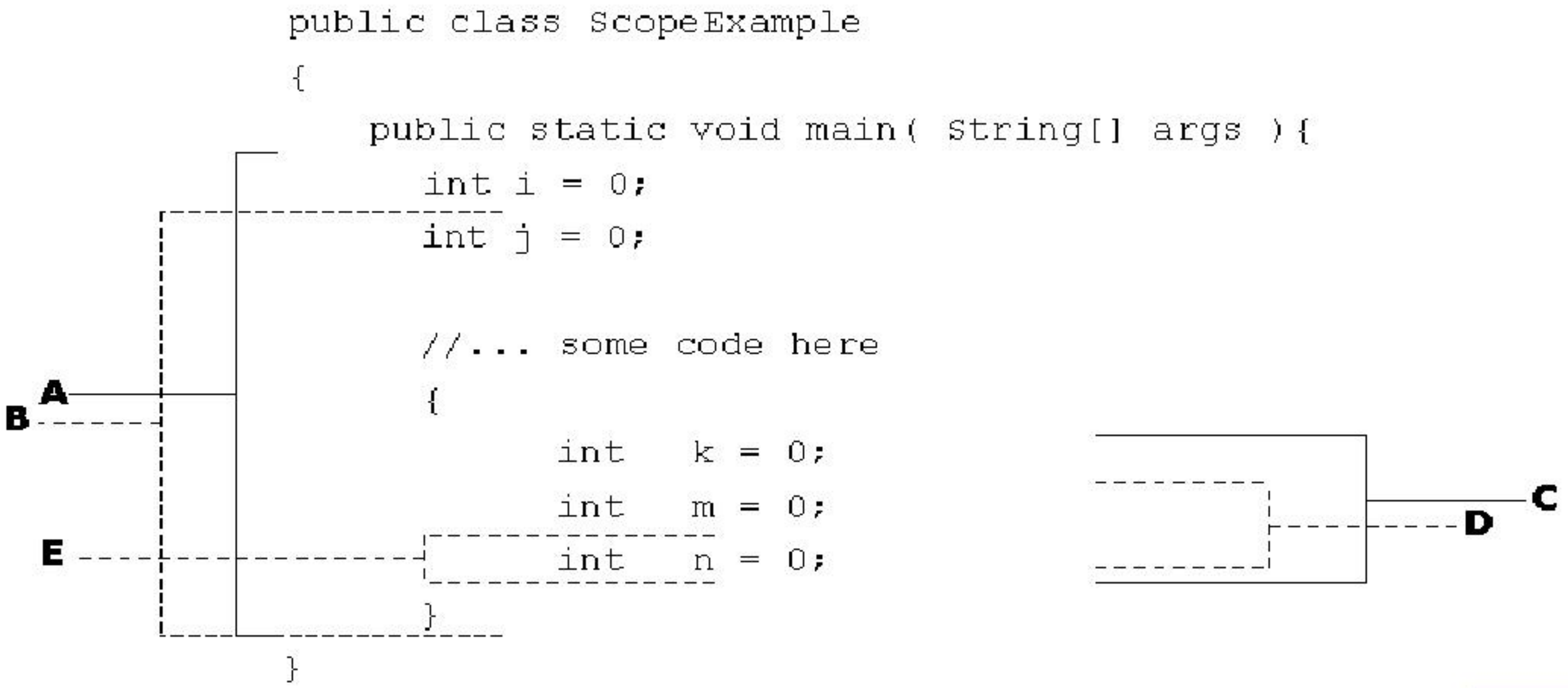
Scope of a Variable

- The scope
  - determines where in the program the variable is accessible.
  - determines the lifetime of a variable or how long the variable can exist in memory.
  - The scope is determined by where the variable declaration is placed in the program.
- To simplify things, just think of the scope as anything between the curly braces {...}. The outer curly braces are called the outer blocks, and the inner curly braces are called inner blocks.

Scope of a Variable

- A variable's scope is
  - inside the block where it is declared, starting from the point where it is declared

Example 1

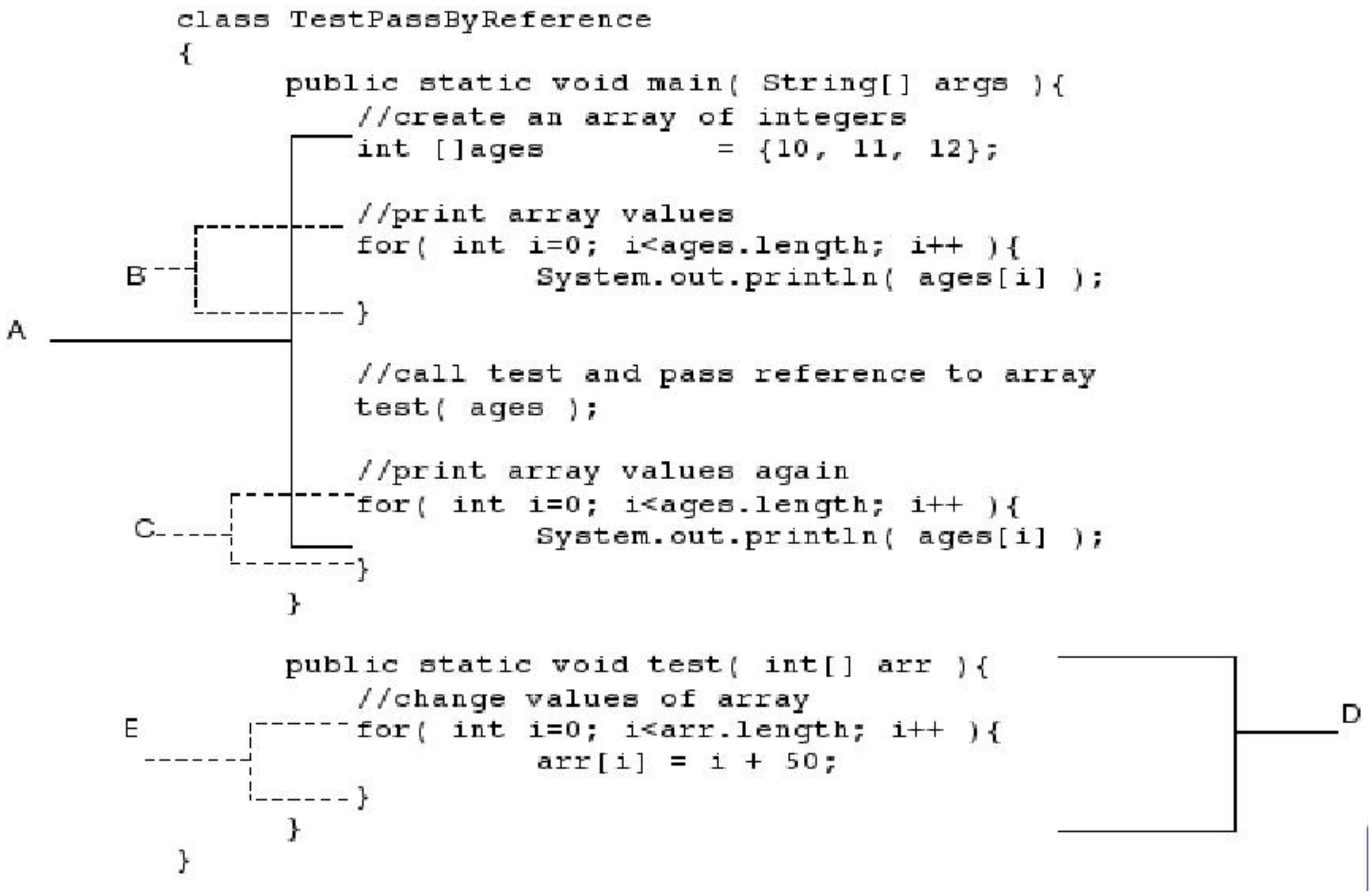


```
public class ScopeExample
{
    public static void main( String[] args ){
        int i = 0;
        int j = 0;

        //... some code here
        {
            int   k = 0;
            int   m = 0;
            int   n = 0;
        }
}
```

Example 1

- The code we have in the previous slide represents five scopes indicated by the lines and the letters representing the scope.
- Given the variables i,j,k,m and n, and the five scopes A,B,C,D and E, we have the following scopes for each variable:
  - The scope of variable i is A.
  - The scope of variable j is B.
  - The scope of variable k is C.
  - The scope of variable m is D.
  - The scope of variable n is E.

Example 2



```
class TestPassByReference
{
        public static void main( String[] args ){
            //create an array of integers
            int []ages          = {10, 11, 12};

            //print array values
            for( int i=0; i<ages.length; i++ ){
                    System.out.println( ages[i] );
            }

            //call test and pass reference to array
            test( ages );

            //print array values again
            for( int i=0; i<ages.length; i++ ){
                    System.out.println( ages[i] );
            }
        }

        public static void test( int[] arr ){
            //change values of array
            for( int i=0; i<arr.length; i++ ){
                    arr[i] = i + 50;
            }
        }
}
```

Example 2

- In the main method, the scopes of the variables are,
  - ages[] - scope A
  - i in B - scope B
  - i in C – scope C

- In the test method, the scopes of the variables are,
  - arr[] - scope D
  - i in E - scope E

Scope of a Variable

- When declaring variables, only one variable with a given identifier or name can be declared in a scope.
- That means that if you have the following declaration,

```
{
      int  test  =  10;
      int  test  =  20;
}
```

your compiler will generate an error since you should have unique names for your variables in one block.

Scope of a Variable

- However, you can have two variables of the same name, if they are not declared in the same block. For example,

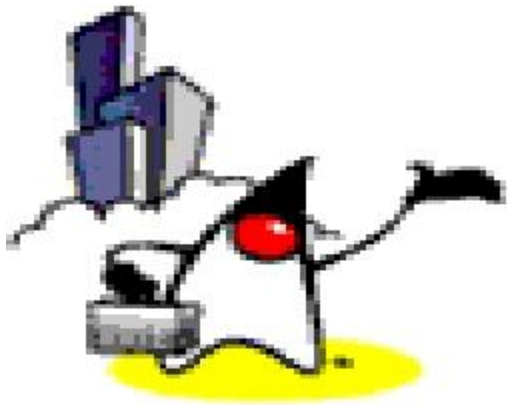
```
int   test   =   0;
System.out.print(   test   );                    //   prints   0

//..some   code   here
if   (   x   ==   2)   {
      int   test   =   20;
      System.out.print(   test   );//   prints   20
}

System.out.print(   test   );                    //   prints   0
```

Scope of Variables

- Local (automatic) variable
  - only valid from the line they are declared on until the closing curly brace of the method or code block within which they are declared
  - most limited scope
- Instance variable
  - valid as long as the object instance is alive
- Class (static) variable
  - in scope from the point the class is loaded into the JVM until the the class is unloaded. Class are loaded into the JVM the first time the class is referenced

Coding Guidelines

- Avoid having variables of the same name declared inside one method to avoid confusion.



Type Casting

Type Casting

- Type Casting
  - Mapping type of an object to another
- To be discussed
  - Casting data with primitive types
  - Casting objects

Casting Primitive Types

- Casting between primitive types enables you to convert the value of one data from one type to another primitive type.
- Commonly occurs between numeric types.
- There is one primitive data type that we cannot do casting though, and that is the boolean data type.
- Types of Casting:
  - Implicit Casting
  - Explicit Casting

Implicit Casting

- Suppose we want to store a value of int data type to a variable of data type double.

```
int            numInt  =  10;
double         numDouble  =  numInt;  //implicit  cast
```

In this example, since the destination variable's data type (double) holds a larger value than the value's data type (int), the data is implicitly casted to the destination variable's data type double.

Implicit Casting

- Another example:

```
int         numInt1  =  1;
int         numInt2  =  2;

//result  is  implicitly  casted  to  type  double
double      numDouble  =  numInt1/numInt2;
```

Explicit Casting

- When we convert a data that has a large type to a smaller type, we must use an explicit cast.

- Explicit casts take the following form:

  (Type)value

  where,

  Type - is the name of the type you're converting to
  value -is an expression that results in the value of the source type

Explicit Casting Examples

```
double        valDouble  =  10.12;
int           valInt  =  (int)valDouble;

//convert  valDouble  to  int  type
double  x  =  10.2;
int  y  =  2;

int  result  =  (int)(x/y);  //typecast  result  of  operation  to  int
```

Casting Objects

- Instances of classes also can be cast into instances of other classes, with one restriction: The source and destination classes must be related by inheritance; one class must be a subclass of the other.
  - We'll cover more about inheritance later.
- Casting objects is analogous to converting a primitive value to a larger type, some objects might not need to be cast explicitly.

Casting Objects

- To cast,

  (classname)object

  where,

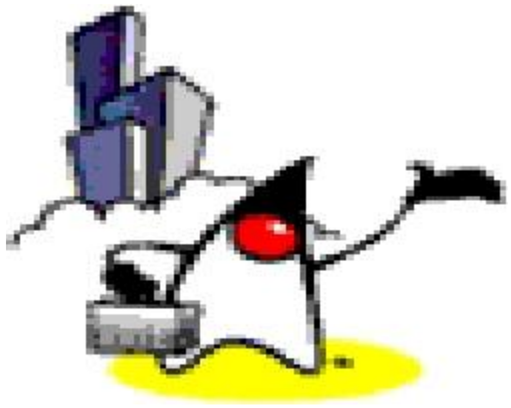
  classname is the name of the destination class &

  object is a reference to the source object

Casting Objects Example

- The following example casts an instance of the class VicePresident to an instance of the class Employee; VicePresident is a subclass of Employee with more information, which here defines that the VicePresident has executive washroom privileges.

```
Employee emp = new Employee();
VicePresident veep = new VicePresident();

// no cast needed for upward use
emp = veep;

// must cast explicitly
veep = (VicePresident)emp;
```



Primitives & Wrapper Types

Converting Primitive types to Objects and vice versa

- One thing you can't do under any circumstance is cast from an object to a primitive data type, or vice versa.

- As an alternative, the java.lang package includes classes that correspond to each primitive data type: Float, Boolean, Byte, and so on. We call them Wrapper classes.

Wrapper Classes

- Most of these classes have the same names as the data types, except that the class names begin with a capital letter
  - Integer is a wrapper class of the primitive int
  - Double is a wrapper class of the primitive double
  - Long is a wrapper class of the primitive long
- Using the classes that correspond to each primitive type, you can create an object that holds the same value.

Converting Primitive types to Objects (Wrapper) and vice versa

- The following statement creates an instance of the Integer class with the integer value 7801

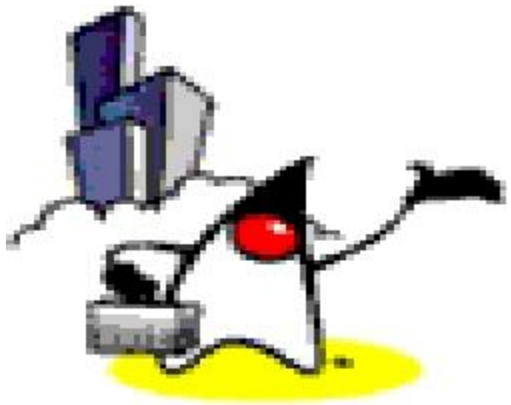
```
Integer dataCount = new Integer(7801);
```

- The following statement converts an Integer object to its primitive data type int. The result is an int with value 7801

```
int newCount = dataCount.intValue();
```

- A common translation you need in programs is converting a String to a numeric type, such as an int (Object->primitive)

```
String pennsylvania = "65000";

int penn = Integer.parseInt(pennsylvania);
```



Comparing Objects

Comparing Objects

- In our previous discussions, we learned about operators for comparing values—equal, not equal, less than, and so on. Most of these operators work only on primitive types, not on objects.
- The exceptions to this rule are the operators for equality: == (equal) and != (not equal). When applied to objects, these operators don't do what you might first expect. Instead of checking whether one object has the same value as the other object, they determine whether both sides of the operator refer to the same object.

Comparing Objects

- Example:

```
class  EqualsTest
{
        public  static  void  main(String[]  arguments)  {
            String  str1,  str2;
            str1   =  "Free  the   bound  periodicals.";
            str2   =  str1;
            System.out.println("String1:  "   +   str1);
            System.out.println("String2:  "   +   str2);
            System.out.println("Same  object?  "  +   (str1  ==   str2));
            str2   =  new  String(str1);
            System.out.println("String1:  "   +   str1);
            System.out.println("String2:  "   +   str2);
            System.out.println("Same  object?  "  +   (str1  ==   str2));
            System.out.println("Same  value?  "  +   str1.equals(str2));
        }
}
```

Comparing Objects

- This program's output is as follows:

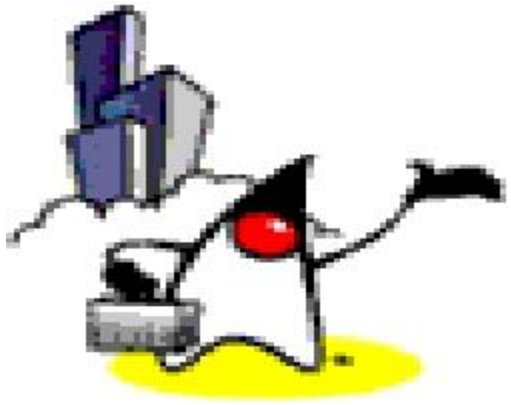
```
String1:  Free  the  bound        periodicals.
String2:  Free  the  bound        periodicals.
Same  object?  true
String1:  Free  the  bound        periodicals.
String2:  Free  the  bound        periodicals.
Same  object?  false
Same  value?  True
```

Comparing Objects

- NOTE on Strings:
  - Given the code:

```
String str1 = “Hello”;
String str2 = “Hello”;
```

  - These two references str1 and str2 will point to the same object.
  - String literals are optimized in Java; if you create a string using a literal and then use another literal with the same characters, Java knows enough to give you the first String object back.
  - Both strings are the same objects; you have to go out of your way to create two separate objects.



getClass() method & instanceof Operator

Determining the class of an object

- Want to find out what an object's class is? Here's the way to do it.

- Suppose we have the following object:

```
SomeClassName  key   =  new   SomeClassName();
```

Now, we'll discuss two ways to know the type of the object pointed to by the variable **key**.

getClass() method

- The getClass() method returns a Class object instance (where Class is itself a class)
- Class class has a method called getName().
  - getName() returns a string representing the name of the class.
- For Example,

```
String   name   =   key.getClass().getName();
```

instanceof operator

- The instanceof has two operands: a reference to an object on the left and a class name on the right.

- The expression returns true or false based on whether the object is an instance of the named class or any of that class's subclasses.

- For Example,

```
boolean  ex1  =  "Texas"  instanceof  String;  //  true
Object  pt  =  new  Point(10,  10);
boolean  ex2  =  pt  instanceof  String;  //  false
```