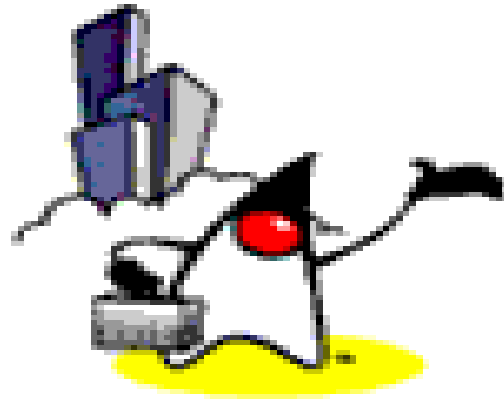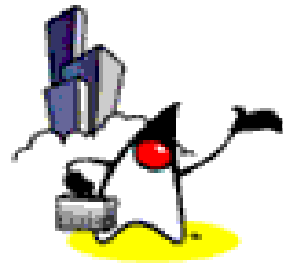# Working with Java Classes

Objectives

At the end of the lesson, the student should be able to:

- Explain object-oriented programming and some of its concepts

- Differentiate between classes and objects

- Differentiate between instance variables/methods and class(static) variables/methods

- Explain what methods are and how to call and pass parameters to methods

- Identify the scope of a variable

- Cast primitive data types and objects

- Compare objects and determine the class of an objects

JEDI

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.

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

JEDI

Encapsulation

- Encapsulation
  - The scheme of hiding implementation details of a class.
  - The caller of the class does not need to know the implementation details of a class
  - The implementation can change without affecting the caller of the class
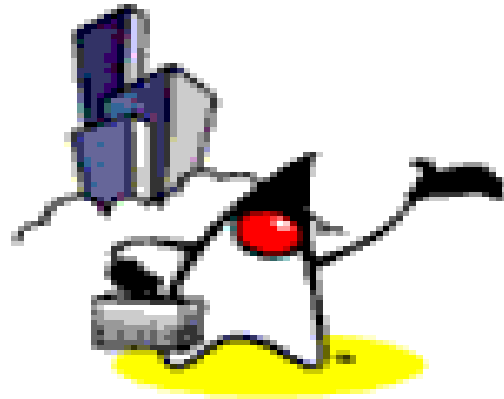
Classes and Objects (Object Instances)

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

JEDI

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.
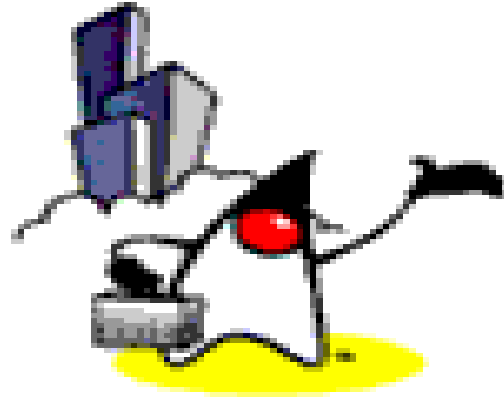
# Creation of Object Instances with "new" keyword

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


```
str2 ─────────> Hello world!
reference to a      String object
String object
```
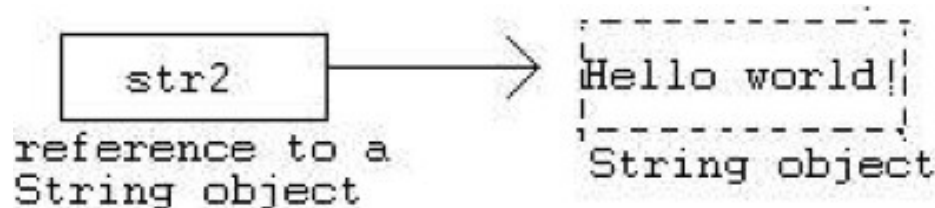
Creation of Object Instance

- 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.

JEDI
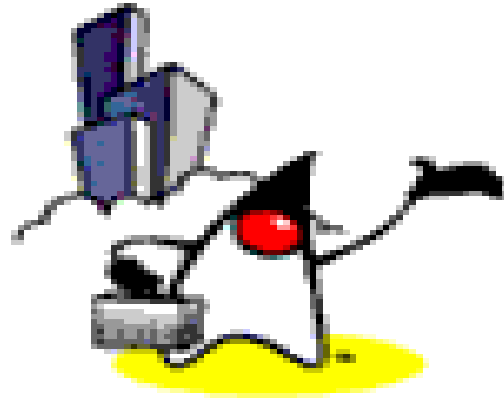
Methods
(Instance methods & Static methods)

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.

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]

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);
```
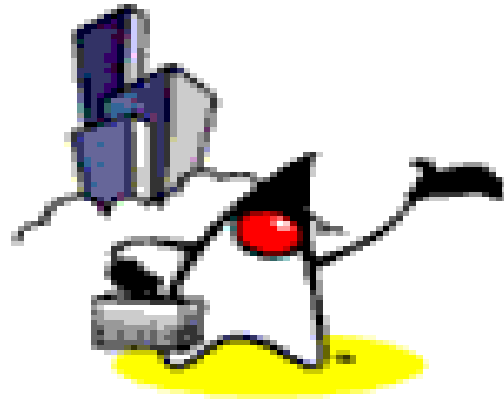
Calling Static Methods

- 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.

JEDI

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 );                                    Pass i as parameter
                                                    which is copied to j

      //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;
    }
}
```

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

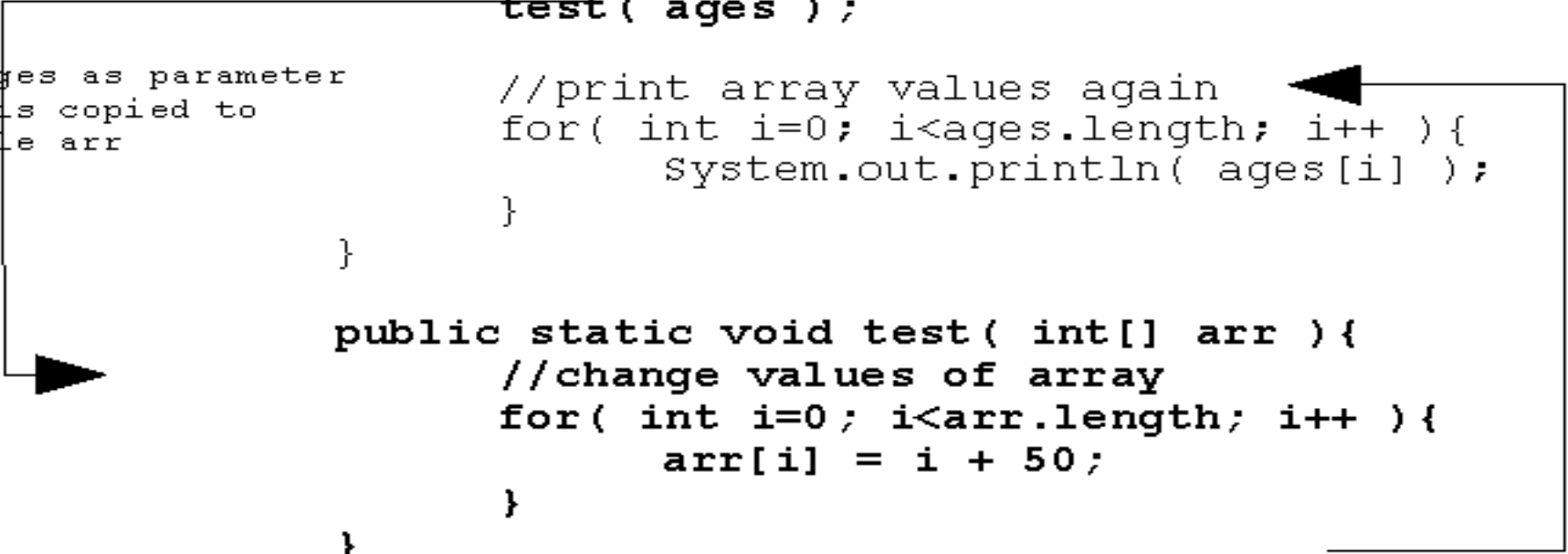```
class TestPassByReference
{
        public static void main(String[] args){
                //create an array of integers
                int []ages = {10, 11, 12};

                //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;
                }
        }
}
```
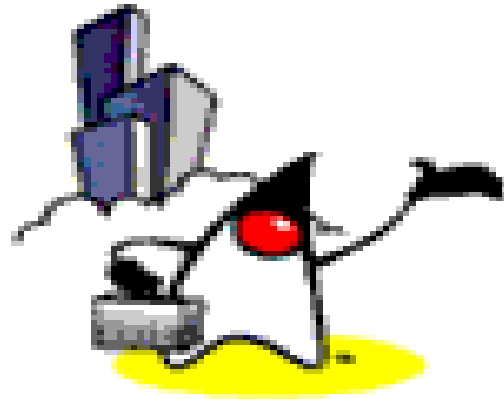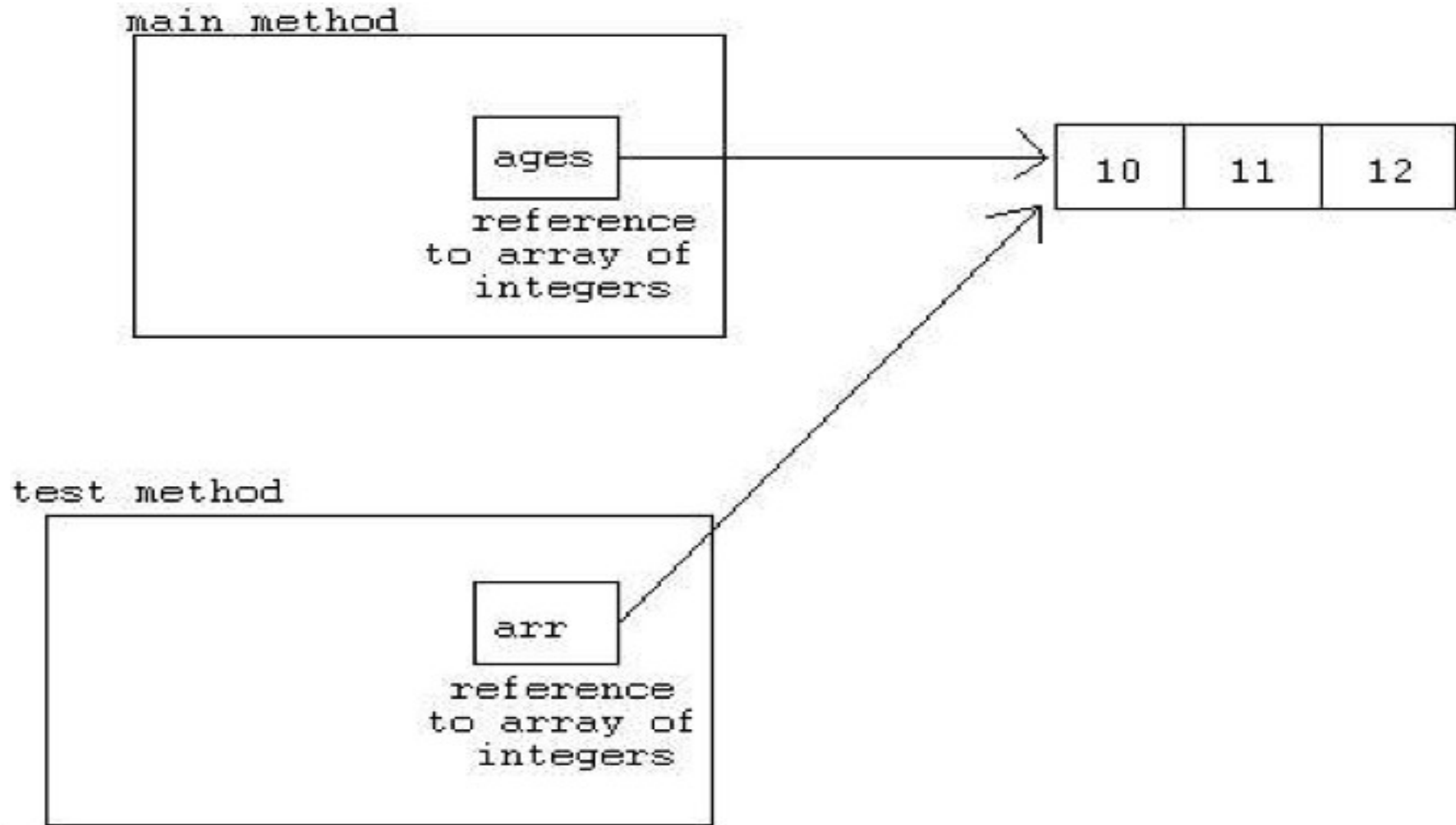
Pass ages as parameter
which is copied to
variable arr

# Pass-by-Reference

# **Variables (Fields, Properties)**

# 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
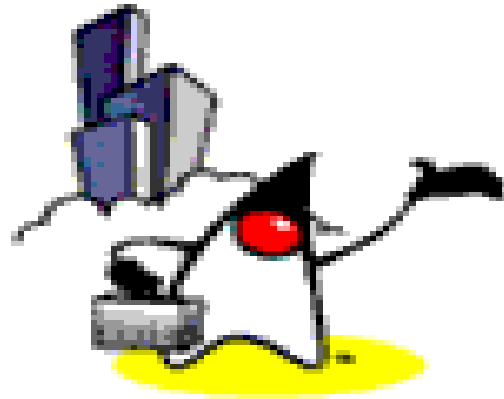
# **Scope of a Variable**

# 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 <span style="color:red">where</span> 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

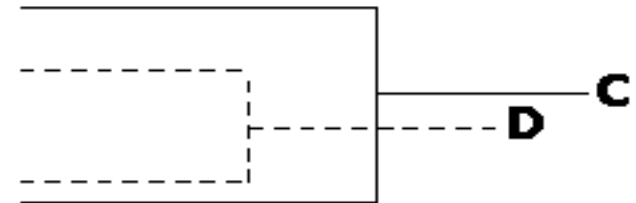# Example 1

```
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;
        }
    }
}
```

A

B

C

D

E

Example 1

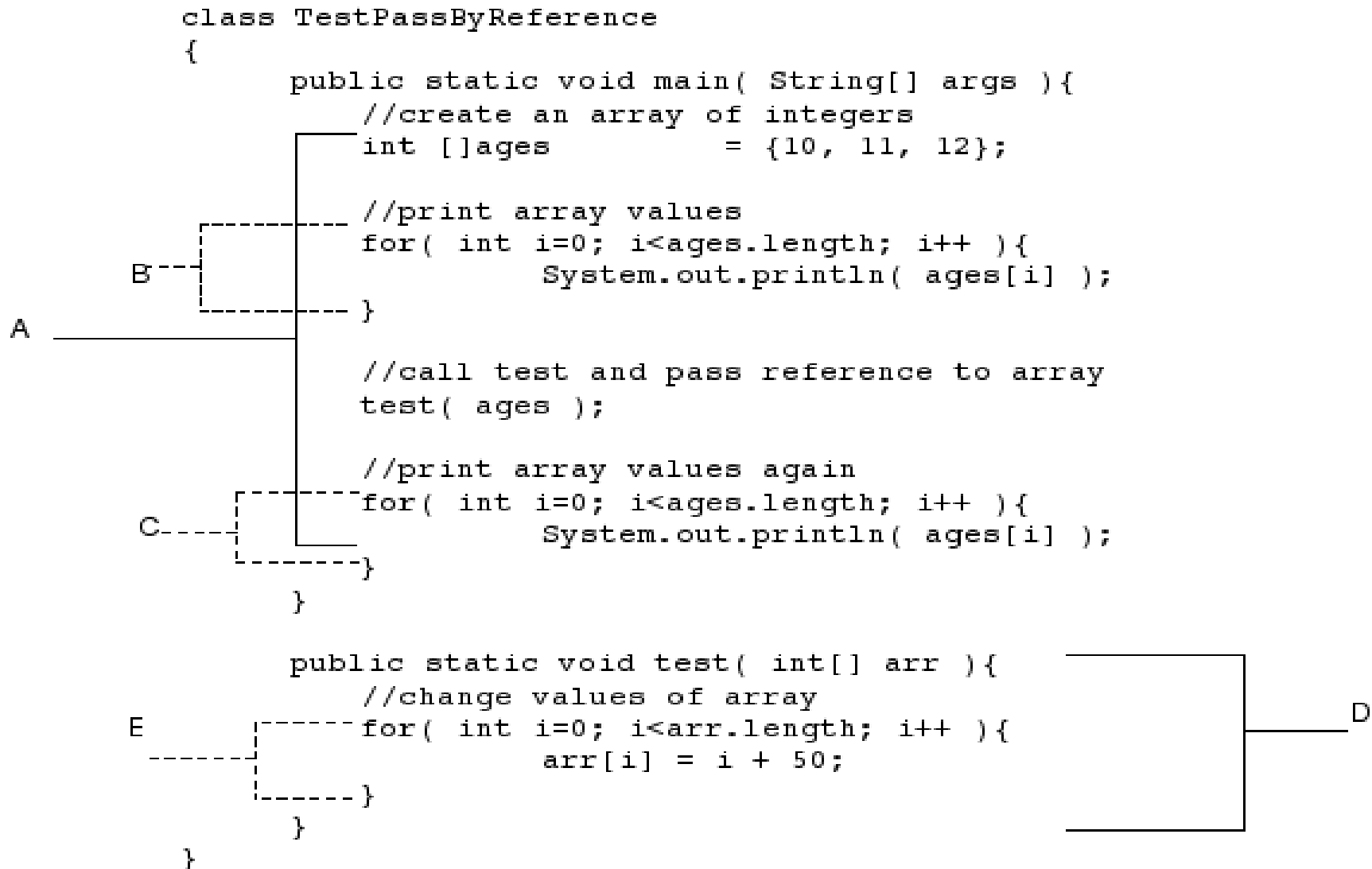- The code we have in the previous slide represents five scopes indicated by the lines and the letters representing the scope.

- 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;
        }
    }
}
```

A

B

C

D

E

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

JEDI

Coding Guidelines

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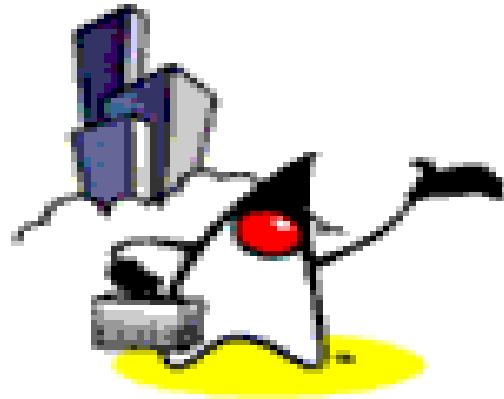
Type Casting

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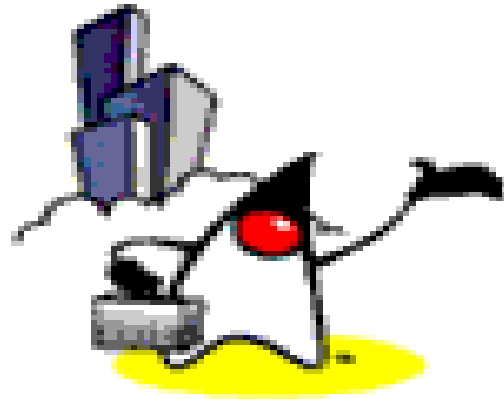
- The following example casts an instance of the class VicePresident to an instance of the class Employee; VicePresident is a subclass of Employee with more information, which here defines that the VicePresident has executive washroom privileges.

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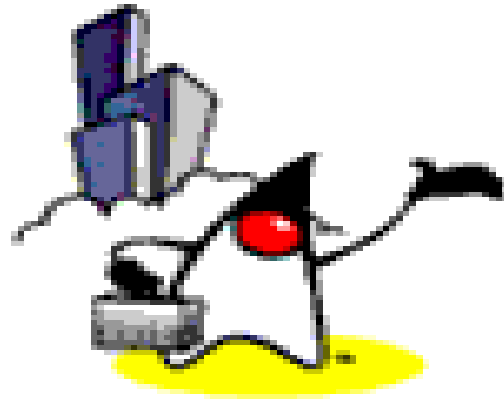
  ```
  int newCount = dataCount.intValue();
  ```

- 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:

```
1   class EqualsTest
2   {
3       public static void main(String[] arguments) {
4           String str1, str2;
5           str1 = "Free the bound periodicals.";
6           str2 = str1;
7           System.out.println("String1: " + str1);
8           System.out.println("String2: " + str2);
9           System.out.println("Same object? " + (str1 == str2));
10          str2 = new String(str1);
11          System.out.println("String1: " + str1);
12          System.out.println("String2: " + str2);
13          System.out.println("Same object? " + (str1 == str2));
14          System.out.println("Same value? " + str1.equals(str2));
15      }
16  }
```

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.
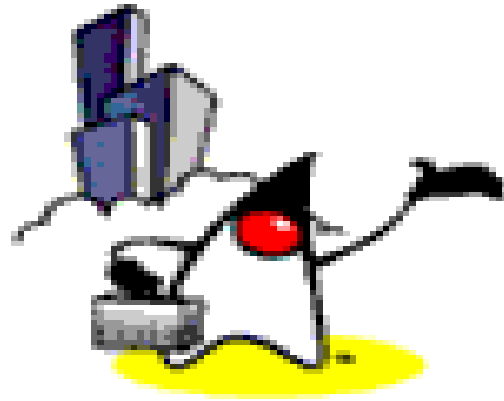
getClass() method & instanceof Operator

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.

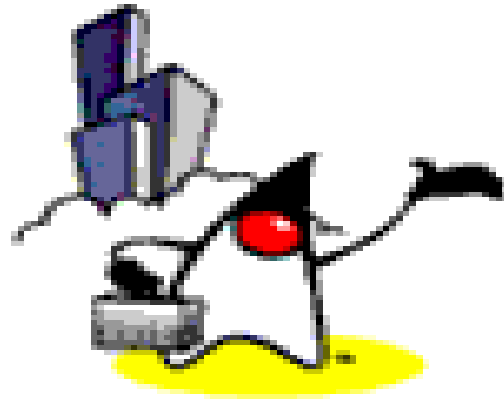- For Example,

```
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
```

Summary

Summary

- Classes and Objects
  - Instance variables
  - Class Variables
- Class Instantiation
- Methods
  - Instance methods
  - Passing Variables in Methods (Pass-by-value, Pass-by-reference)
  - Static methods
- Scope of a variable
- Casting (object, primitive types)
- Converting Primitive Types to Objects and Vice Versa
- Comparing Objects
- Determining the Class of an Object