

Day 1

Introduction to Java

Objectives

-  At the end of the lesson, the student should be able to:
- Describe the features of Java technology such as the Java virtual machine, garbage collection and code security
 - Describe the different phases of a Java program

Java Background: History



Java

- Was created in 1991
- By James Gosling et al. of Sun Microsystems.
- Initially called Oak, in honor of the tree outside Gosling's window,
- its name was changed to Java because there was already a language called Oak.

What is Java Technology?

The Java technology is:

- A programming language
- A development environment
- An application environment
- A deployment environment

Java Features



Some features of Java:

1. The Java Virtual Machine
2. Garbage Collection
3. Code Security

Java Features:

The Java Virtual Machine

Java Virtual Machine (JVM)

- An imaginary machine that is implemented by emulating software on a real machine
- provides the hardware platform specifications to which you compile all Java technology code called ByteCode

Bytecode

- A special machine language that can be understood by the Java Virtual Machine (JVM)
- independent of any particular computer hardware, so any computer with a Java interpreter can execute the compiled Java program, no matter what type of computer the program was compiled on

Garbage Collection

Garbage collection thread

- Responsible for freeing any memory that can be freed. This happens automatically during the lifetime of the Java program.
- Programmer is freed from the burden of having to deallocate that memory themselves

Code Security

 Code security is attained in Java through the implementation of its Java Runtime Environment (JRE)

JRE

- runs code compiled for a JVM and performs class loading (through the class loader)
- code verification (through the bytecode verifier)
- and finally code execution

Code Security

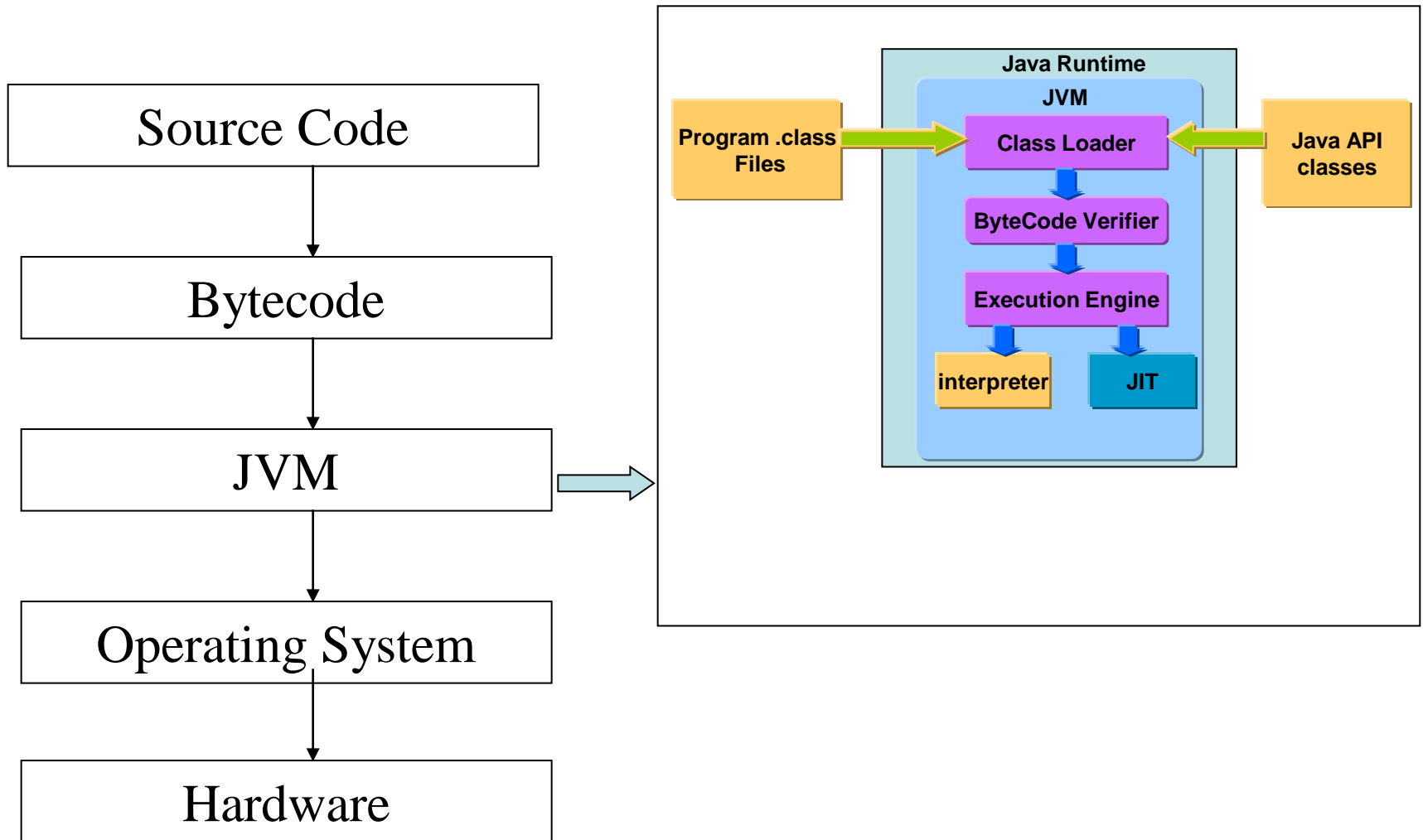
Class Loader

- responsible for loading all classes needed for the Java program
- adds security by separating the namespaces for the classes of the local file system from those that are imported from network sources
- After loading all the classes, the memory layout of the executable is then determined. This adds protection against unauthorized access to restricted areas of the code since the memory layout is determined during runtime

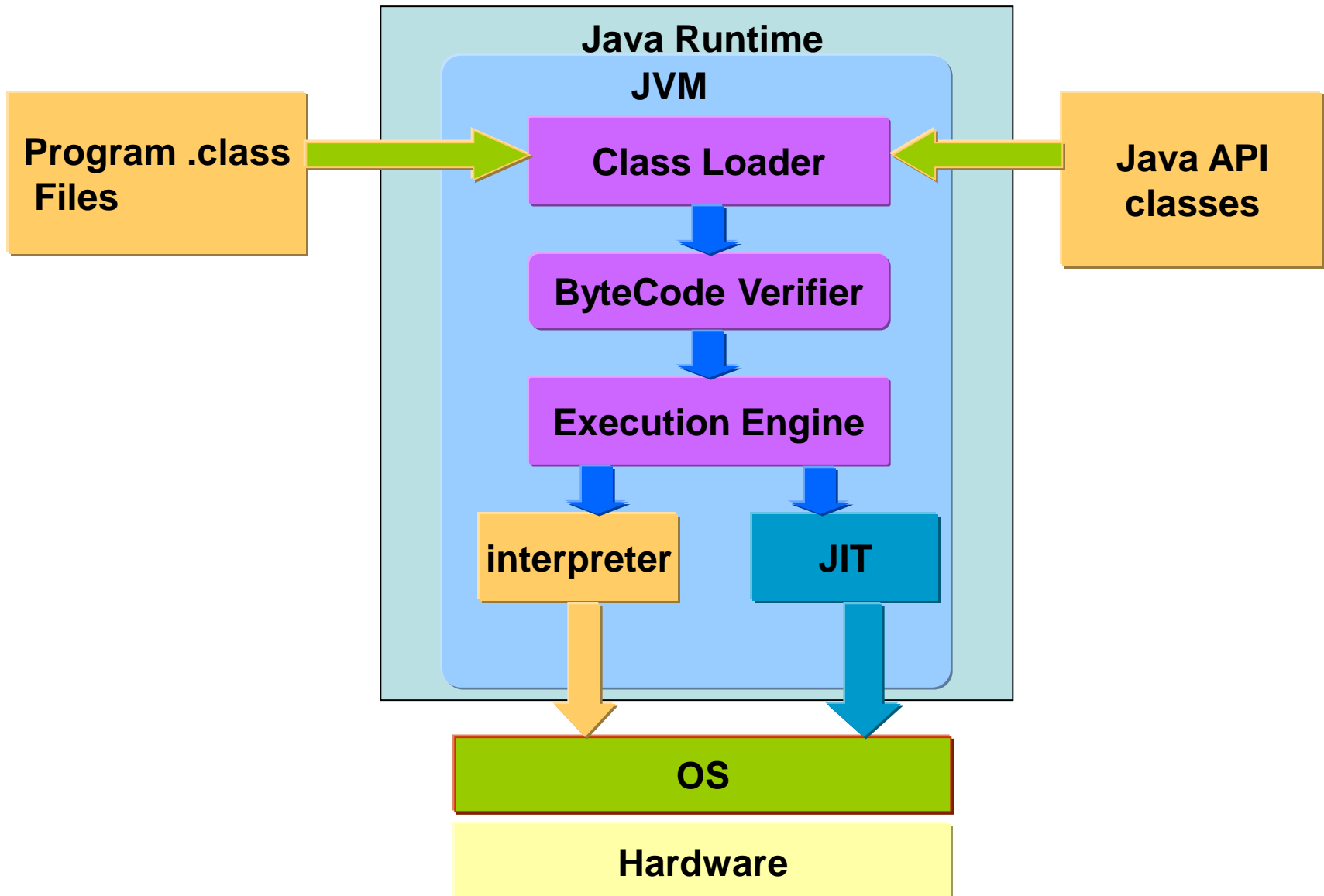
Code Security

- Bytecode verifier tests the format of the code fragments and checks the code fragments for illegal code that can violate access rights to objects

Java Source to Execution Flow



Java Virtual Machine



The Java Buzzwords

Simple

- Small language [large libraries]
- Small interpreter (40 k), but large runtime libraries(175 k)

Object-Oriented

- Supports encapsulation, inheritance, abstraction, and polymorphism.

Distributed

- Libraries for network programming
- Remote Method Invocation

Architecture neutral

- Java Bytecodes are interpreted by the JVM.

The Java Buzzwords

Secure

- Difficult to break Java security mechanisms
- Java Bytecode verification
- Signed Applets.

Portable

- Primitive data type sizes and their arithmetic behavior specified by the language
- Libraries define portable interfaces

Multithreaded

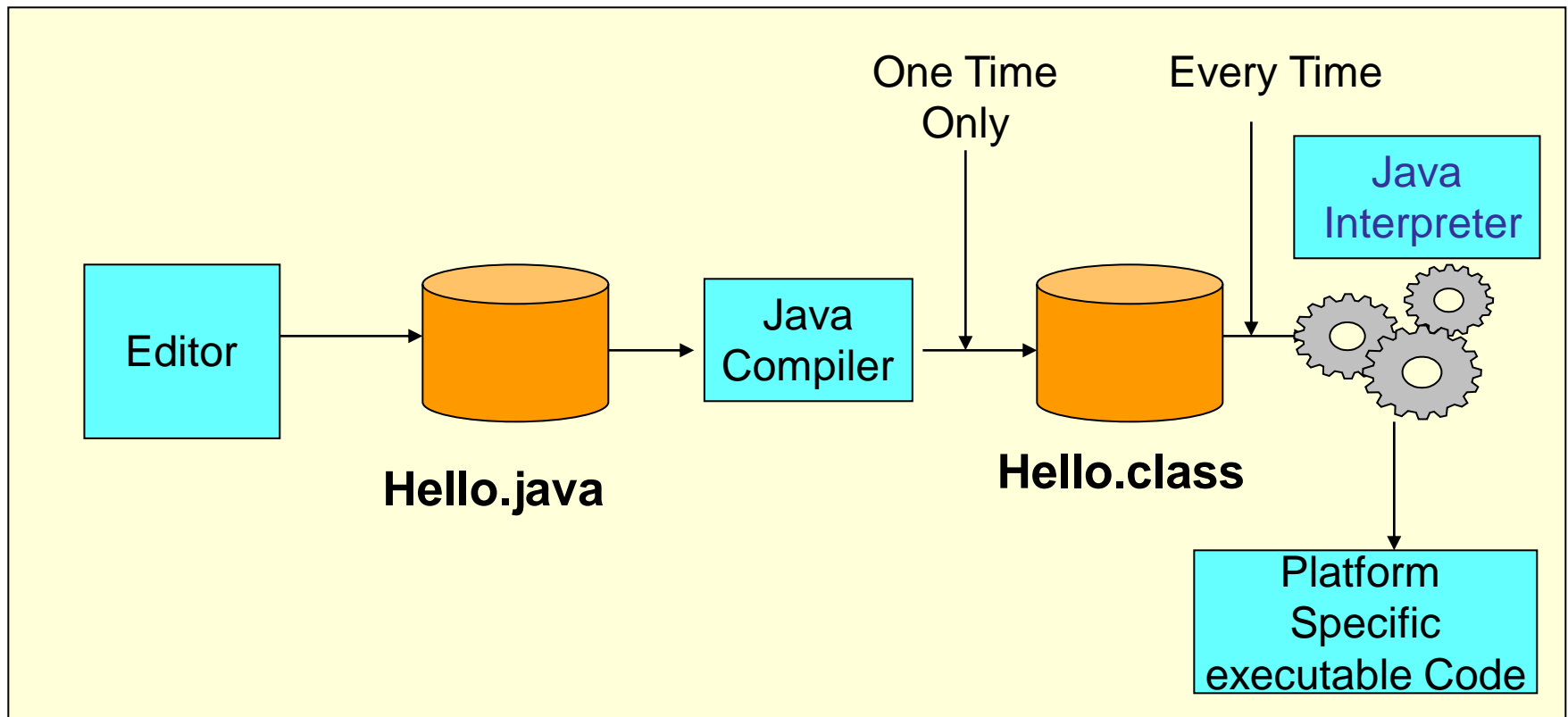
- Threads are easy to create and use

Dynamic

- Finding Runtime Type Information is easy

Phases of a Java Program

- The following figure describes the process of compiling and executing a Java program



Phases of a Java Program

Task	Tool to Use	Output
Write The Program	Any Text Editor	File with .java extension
Compile The Program	Java Compiler (javac)	File with .class extension (ByteCode)
Run the Program	Java Interpreter (java)	Program Output

Know your Programming Environment

Definitions

Console

- This is where you type in commands
- Examples are Terminal (Linux), MSDOS Command Prompt (Windows)

Definitions

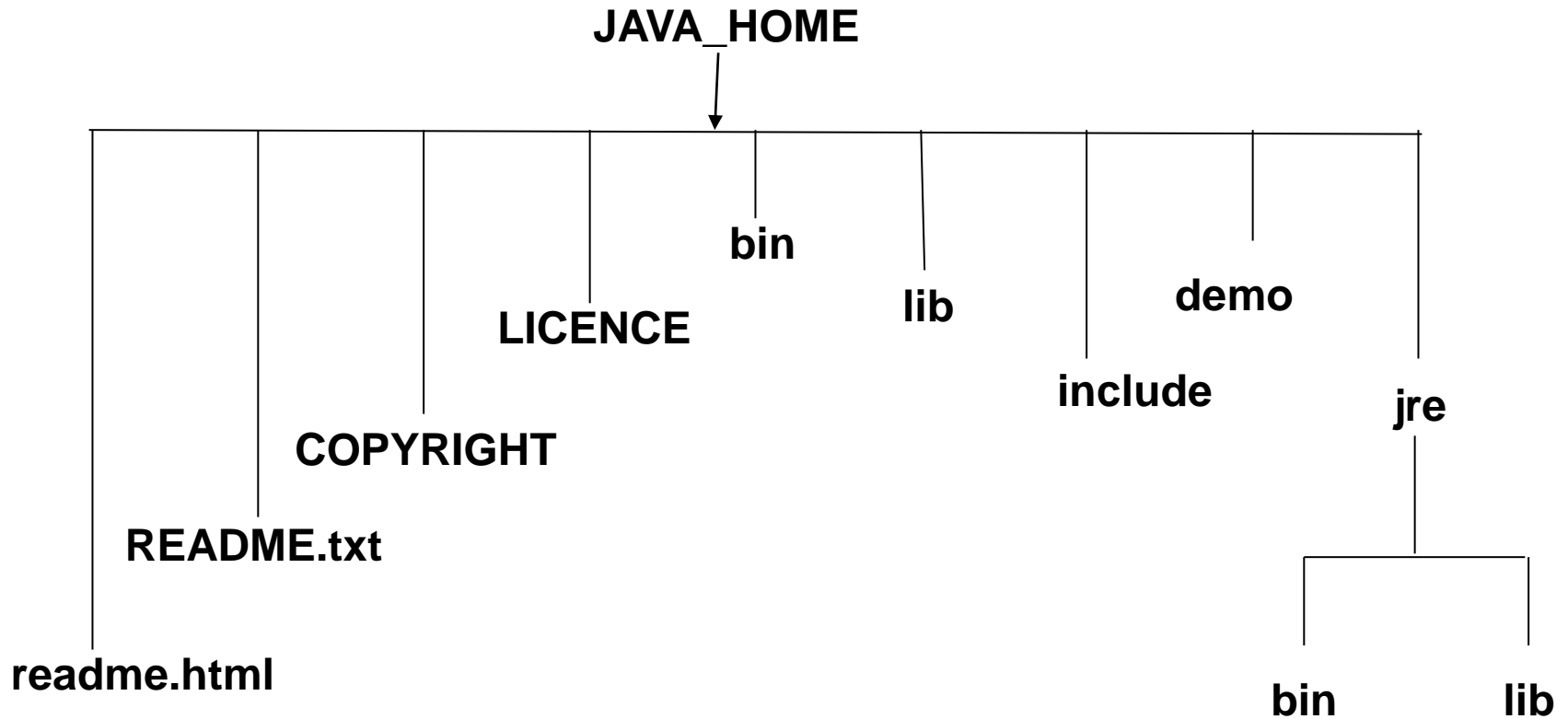
Text Editor

- Examples: Notepad, Wordpad, Vi

Definitions

- Integrated Development Environment or IDE
 - a programming environment integrated into a software application that provides a GUI builder, a text or code editor, a compiler and/or interpreter and a debugger.

Installation and Setup JDK



Path and Classpath

- PATH is an OS environment variable to locate the executables
- The classpath is a user defined environment variable used by Java to determine where predefined classes are located.
- The Demo will show you how to set PATH to set up Java

Our First Java Program

```
1. public class Hello {  
2.     /**  
3.     * My first Java program  
4.     */  
5.     public static void main( String[] args ){  
  
6.         //prints the string "Hello world" on screen  
7.         System.out.println("Hello world");  
8.     }  
9. }
```


Demo

- Create a Simple Java Application
- Compile the Application using OS terminal/Command prompt
- Run the Application from Command Prompt/Terminal

Programming Fundamentals

Introduction to OO Programming

Classes and Objects

Class

- can be thought of as a template, a prototype or a blueprint of an object
- A class is a structure that defines the data and the methods to work on that data.
- is the fundamental structure in object-oriented programming
- is a user defined data type

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
 - An instance is an executable copy of a class.
- There can be any number of objects of a given class in memory at any one time.
- 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

A blue print of a person

Person class

Name

Age

displayDetails()

Real Time Entities

**Jenifer
32**



**Peter
45**



**Suraj Bhan
25**



**Kirti
23 years**



Object Oriented Programming

Definition

“ Object oriented programming is method of implementation in which programs are organized as collection of objects ,each of which represent an instance of class”

“Object-oriented programming is a method of programming based on a hierarchy of classes, and well-defined and cooperating objects which communicate with each other by passing messages”

Object Oriented Programming

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.

Object Oriented Programming

Example of objects

Object	Properties	Behavior
Car	type of transmission manufacturer color	turning, braking accelarating
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**

OOP Concepts

■ There are three major concepts in object-oriented programming:

- encapsulation
- inheritance
- polymorphism.

■ We use encapsulation, Inheritance and polymorphism in Object Oriented Design to achieve ABSTRACTION

Encapsulation

Encapsulation

- The scheme of hiding implementation details of a class.
- Encapsulation refers to the creation of self-contained modules that bind processing functions to the data
- 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
- **Encapsulation Enforces Modularity**

Inheritance

Inheritance

- transfer of properties from the parent class to the child class
- Classes are created in hierarchies, and inheritance allows the structure and methods in one class to be passed down the hierarchy
- Inheritance facilitates reusability of code (objects, methods, data)
- The ability to reuse existing objects is considered a major advantage of object oriented technology.

“Inheritance Passes "Knowledge" Down”

Polymorphism

- polymorphic (poly-> many, morphs-> forms) means “multiple shapes of same thing”
 - In the context of OOP polymorphic means “having multiple behavior”
 - most commonly polymorphic word is used for same named method having different implementations
 - a polymorphic method results in the different actions depending upon the object being referenced
- “Polymorphism Takes any Shape”**

Where do we use the OOP in Java

- As we progress in java programming, we explain in each step where do we use OOP Concepts.

Java Statements

Statement

- one or more lines of code terminated by a semicolon.
- Example:
 - `System.out.println("Hello world");`

Java Blocks

Block

- is one or more statements bounded by an opening and closing curly braces that groups the statements as one unit.
- Block statements can be nested indefinitely.
- Any amount of white space is allowed.

Example:

```
public static void main( String[] args ){  
    System.out.println("Hello");  
    System.out.println("world");  
}
```


Java Identifiers

Identifiers

- are tokens that represent names of variables, methods, classes, etc.
- Examples of identifiers are: Hello, main, System, out.

Java identifiers are case-sensitive.

- This means that the identifier **Hello** is not the same as **hello**.

Java Identifiers

- Identifiers must begin with either a letter, an underscore “_”, or a dollar sign “\$”.
- Letters may be lower or upper case.
- Subsequent characters may use numbers 0 to 9.
- Identifiers cannot use Java keywords like class, public, void, etc. We will discuss more about Java keywords later.

Java Identifiers Coding Guidelines

■ For names of classes, capitalize the first letter of the class name.

■ For example,
`ThisIsAnExampleOfClassName`

■ For names of methods and variables, the first letter of the word should start with a small letter. For example,
`thisIsAnExampleOfMethodName`

Java Identifiers Coding Guidelines

- In case of multi-word identifiers, use capital letters to indicate the start of the word except the first word. For example,
`charArray, fileNumber, ClassName.`
- Avoid using underscores at the start of the identifier such as `_read` or `_write`.

Java Keywords

- Keywords are predefined identifiers reserved by Java for a specific purpose.
- You cannot use keywords as names for your variables, classes, methods ... etc.
- The next slide contains the list of the Java Keywords.

Java Keywords

abstract	const	finally	int	public	this
boolean	continue	float	interface	return	throw
break	default	for	long	short	throws
byte	do	goto	native	static	transient
case	double	if	new	strictfp	try
catch	else	implements	package	super	void
char	extends	import	private	switch	volatile
class	final	instanceof	protected	synchronized	While
native	enum				



reserved words but not used in java

Variables

- A variable is an item of data used to store the state of objects.

- A variable has a:
 - data type
 - The data type indicates the type of value that the variable can hold.
 - name

- The variable name must follow rules for identifiers.


Declaring and Initializing Variables

 Declare a variable as follows:

`<data type> <name> [=initial value];`

 Example:

`int num = 20;`

 Note: Values enclosed in <> are required values, while those values in [] are optional.

Outputting Variable Data

- In order to output the value of a certain variable, we can use the following commands:

`System.out.println()`

`System.out.print()`

Outputting Variable Data: Sample Program

```
1 public class OutputVariable {  
2     public static void main( String[] args ){  
3         int value = 10;  
4         char x;  
5         x = 'A';  
6  
7         System.out.println( value );  
8         System.out.println( "The value of x=" + x );  
9     }  
10 }
```

The program will output the following text on screen:

10

The value of x=A

System.out.println() vs. System.out.print()

System.out.println()

- Appends a newline at the end of the data output

System.out.print()

- Does not append newline at the end of the data output

Data Types in Java

 The Java programming language has two categories of data types

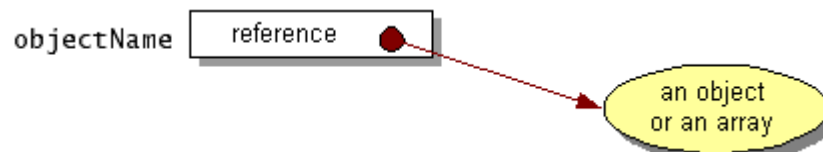
1. primitive
2. reference

Primitive Data Types

Keyword	Description/Size (bytes)	Example
Integer Type		
byte	1	
short	2	
int	4	178
long	8	8864L
Real numbers		
float	4	87.363F
double	8	37.266 or 37.266D
Other Types		
char	2	'c '
boolean	true/false	
void	nothing	

Reference Data Types

- Arrays, classes, and interfaces are reference types.
- The value of a reference type variable is a reference to (an address of) the value or set of values represented by the variable.
- A reference is called a pointer, or a memory address in other languages. The Java programming language does not support the explicit use of addresses like other languages do. You use the variable's name instead.



Reference Variables vs. Primitive Variables

Primitive Variables

- variables with primitive data types such as int or long.
- stores data in the actual memory location of where the variable is

Reference Variables vs. Primitive Variables

Reference Variables

- variables that stores the address in the memory location
- points to another memory location where the actual data is
- When you declare a variable of a certain class, you are actually declaring a reference variable to the object with that certain class.

Example

■ Suppose we have two variables with data types `int` and `String`.

`int num = 10; // primitive type`

`String name = "Hello"; // reference type`

Example

- The picture shown below is the actual memory of your computer, wherein you have the address of the memory cells, the variable name and the data they hold.

Memory Address

Variable Name

Data

1001

int

10

:

:

1790

name

address(2009)

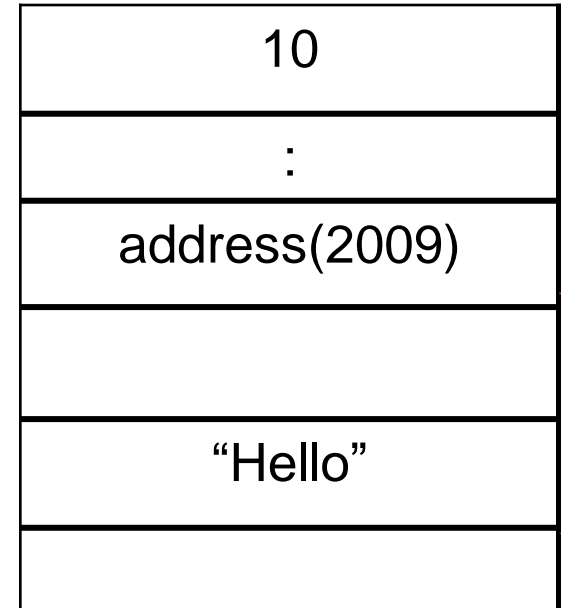
:

:

2009

"Hello"

:



10
:
address(2009)
"Hello"

Operators in Java

Operators

- Different types of operators:
 - arithmetic operators
 - relational operators
 - logical operators
 - Assignment Operators
 - Other Operators
- These operators follow a certain kind of precedence so that the compiler will know which operator to evaluate first in case multiple operators are used in one statement.

Arithmetic Operators

Operator	Use	Description
+	$\text{op1} + \text{op2}$	Adds op1 and op2
-	$\text{op1} - \text{op2}$	Subtracts op2 from op1
*	$\text{op1} * \text{op2}$	Multiplies op1 by op2
/	$\text{op1} / \text{op2}$	Divides op1 by op2
%	$\text{op1} \% \text{op2}$	Computes the remainder of dividing op1 by op2

Increment and Decrement Operators

Operator	Use	Description
++	op++	Increments op by 1; evaluates to the value of op before it was incremented
++	++op	Increments op by 1; evaluates to the value of op after it was incremented
--	op--	Decrements op by 1; evaluates to the value of op before it was decremented
--	--op	Decrements op by 1; evaluates to the value of op after it was decremented

Relational Operators

- Relational operators compare two values and determines the relationship between those values.
- The output of evaluation are the boolean values true or false.

Operator	Use	Returns true if
>	op1 > op2	op1 is greater than op2
>=	op1 >= op2	op1 is greater than or equal to op2
<	op1 < op2	op1 is less than op2
<=	op1 <= op2	op1 is less than or equal to op2
==	op1 == op2	op1 and op2 are equal
!=	op1 != op2	op1 and op2 are not equal

Logical Operators

- Logical operators have one or two boolean operands that yield a boolean result.
- There are six logical operators:
 - `&&` (logical AND)
 - `&` (Bitwise AND)
 - `||` (logical OR)
 - `|` (Bitwise inclusive OR)
 - `^` (Bitwise exclusive OR)
 - `!` (logical NOT)

Logical Operators

■ conditional operators to form multi-part decisions.

Operator	Use	Returns true if
&&	op1 && op2	op1 and op2 are both true, conditionally evaluates op2
	op1 op2	either op1 or op2 is true, conditionally evaluates op2
!	! op	op is false
&	op1 & op2	op1 and op2 are both true, always evaluates op1 and op2
	op1 op2	either op1 or op2 is true, always evaluates op1 and op2
^	op1 ^ op2	if op1 and op2 are different--that is if one or the other of the operands is true but not both

Logical Operators

- The basic expression for a logical operation is, $\text{op1 } \text{op} \text{ op2}$
- where, op1 , op2 - can be boolean expressions, variables or constants
- op - is either $\&\&$, $\&$, $\|$, $|$ or \wedge operator.

Assignment Operators

- The basic assignment operator assigns the value of op2 to op1.

`op1 = op2;`

- In addition to the basic assignment operation, the Java programming language defines these short cut assignment operators that perform an operation and an assignment using one operator. (see next slide)

Assignment Operators

Operator	Use	Equivalent to
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>
<code>&=</code>	<code>op1 &= op2</code>	<code>op1 = op1 & op2</code>
<code> =</code>	<code>op1 = op2</code>	<code>op1 = op1 op2</code>
<code>^=</code>	<code>op1 ^= op2</code>	<code>op1 = op1 ^ op2</code>
<code><<=</code>	<code>op1 <<= op2</code>	<code>op1 = op1 << op2</code>
<code>>>=</code>	<code>op1 >>= op2</code>	<code>op1 = op1 >> op2</code>
<code>>>>=</code>	<code>op1 >>>= op2</code>	<code>op1 = op1 >>> op2</code>

Other Operators

■ The other operators that the Java programming language supports

Operator	Description
?:	Shortcut if-else statement
[]	Used to declare arrays, create arrays, and access array elements
.	Used to form qualified names
(<i>params</i>)	Delimits a comma-separated list of parameters
(<i>type</i>)	Casts (converts) a value to the specified type
new	Creates a new object or a new array
instanceof	Determines whether its first operand is an instance of its second operand

Other Operators

The ?: operator

- The ?: operator is a conditional operator
- ?: is short-hand for an if-else statement:

`op1 ? op2 : op3`

- The ?: operator returns op2 if op1 is true or returns op3 if op1 is false

Other Operators

The [] Operator

- square brackets are used to declare arrays, to create arrays, and to access a particular element in an array.
-
- An example of an array declaration:

```
float[] arrayOfFloats = new float[10];
```

The code declares an array that can hold ten floating point numbers.

an example how to access 7th item in that

```
array: arrayOfFloats[6];
```

Other Operators

The . Operator

- The dot (.) operator accesses instance members of an object or class members of a class

The (type) Operator

- Casts (or "converts") a value to the specified type

e.g.

```
int i=65;
```

```
char ch = (char)i; //converts i into a character 'A'
```


Other Operators

The **new** Operator

- The new operator is used to create a new object or a new array

Here's an example of creating a new Integer object from the Integer class in the ***java.lang package***:

```
Integer anInteger = new Integer(10);
```

- The new Operator allocates memory for the Object or Array

Other Operators

The *instanceof* Operator

- The instanceof operator tests whether its first operand is an instance of its second.

op1 instanceof op2

- op1 must be the name of an object and op2 must be the name of a class
- An object is considered to be an instance of a class if that object directly or indirectly descends from that class

Control Structures

Control Structures



Control structures

- allows us to change the ordering of how the statements in our programs are executed



Two types of Control Structures

1. decision control structures
 - allows us to select specific sections of code to be executed
2. repetition control structures
 - allows us to execute specific sections of the code a number of times

Decision Control Structures

Decision control structures

- Java statements that allows us to select and execute specific blocks of code while skipping other sections

Types:

- if-statement
- if-else-statement
- If-else if-statement

Repetition Control Structures

Repetition control structures

- are Java statements that allows us to execute specific blocks of code a number of times.

Types:

- while-loop
- do-while loop
- for-loop

Branching Statements

- Branching statements allows us to redirect the flow of program execution.
- Java offers three branching statements:
 - `break`
 - `continue`
 - `return.`

Arrays In Java

- Arrays Store multiple values of the same type in a single variable.
- In Java Arrays are Objects and created using the “new” operator

```
Int[] nums=new int[5] OR Int nums[]=new int[5]
```

```
String[] args=new String[10]
```

OR

```
String args[]=new String[10]
```


int array

0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---

indeces

0 1 2 3 4 5 6 7 8 9

Data Population in Arrays

 When you create the array without initial values, all the locations will have default values based on the data type as shown

int array

0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---

indices

0 1 2 3 4 5 6 7 8 9

String array

null	null	null	null	null	null	null	null	null	null
------	------	------	------	------	------	------	------	------	------

indices

0 1 2 3 4 5 6 7 8 9

Day 2

Working with Classes and Objects

Variables in a class

 A class can contain two types of variables

1. Instance variables
2. Class Variables

Instance Variables and Class 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
 - declared with “static” keyword
 - e.g. `static int i=30;`
- This means that they have the same value for all the object instances in the same class
- class variables are similar to “Global variables in other languages”

Class Variables

🌐 For example,

Car class		Object Car A	Object Car B
Instance Variables	Plate Number	ABC 111	XYZ456
	Color	Red	Blue
	Manufacturer	Honda	Toyota
	Current Speed	120 km/hr	120 km/hr
class Variable	carCount=2		
methods	Accelerate		
	Turn		
	Apply brake		

Creation of Object Instance

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 special method where you place all the initializations, it has the same name as the class.

Methods

Methods

Method

- is a separate piece of code with a **name** 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.

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.

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

Mutator (Setter) Methods

Mutator Methods

- used to write or change values of our class variables (instance/static).
- Usually written as:
- **set<NameOfInstanceVariable>**

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

Parameter Passing

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

```
class TestPassByValue {  
    public static void main(String[] args)  
    {  
        int i=10;  
        //print the value of i  
        System.out.println(i);  
        //call the method test() and pass i to the method test  
        test(i);  
  
        //print the value of i. i not changed  
        System.out.println(i);  
    }  
    public static void test(int j) {  
        //change the value of parameter j  
        j=30;  
    }  
}
```

pass i as parameter
which is copied to j

Parameter Passing

Pass-by-Reference

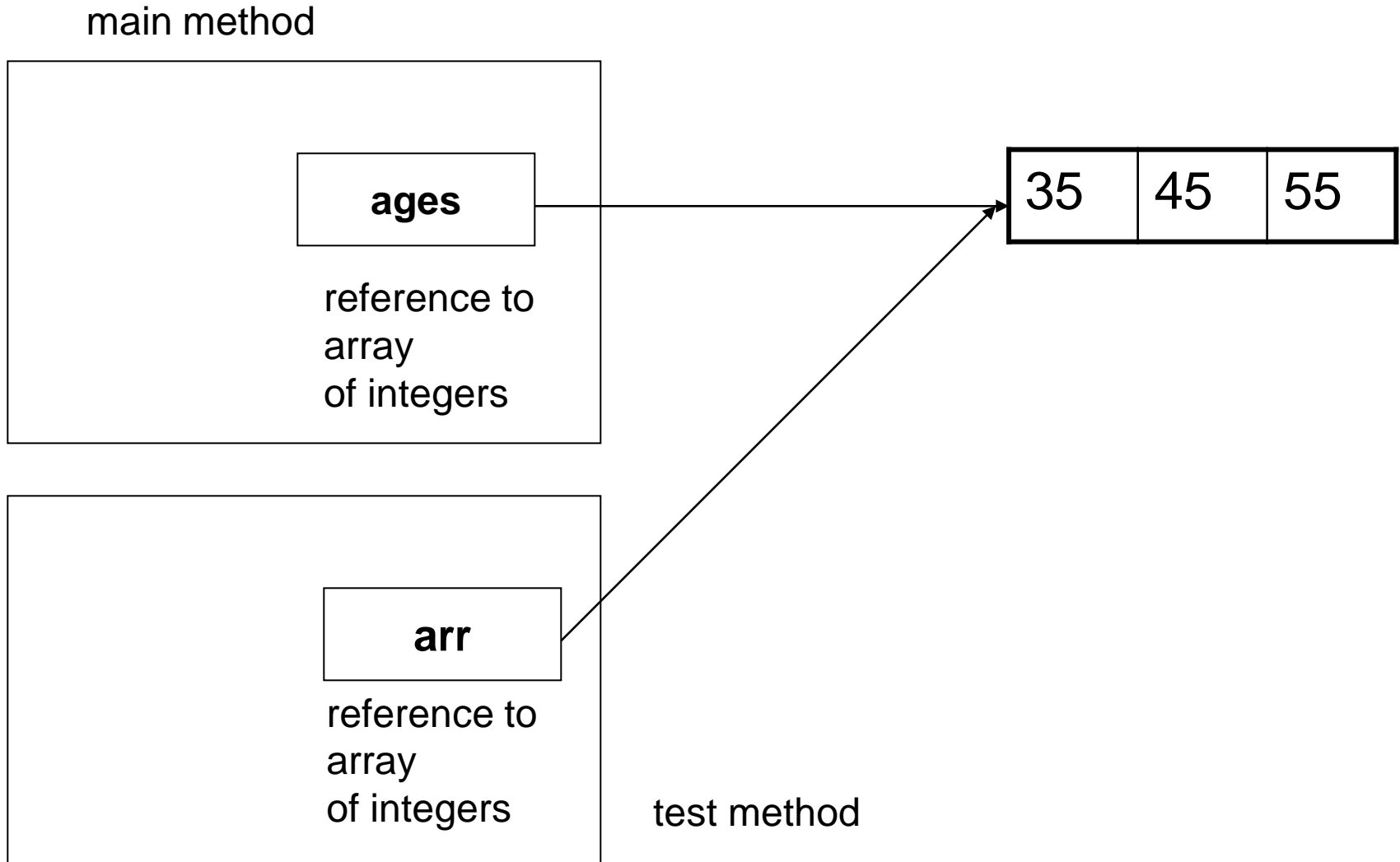
- When a pass-by-reference occurs, the reference to an object is passed to the calling method. This means that, the method makes a copy of the reference of the variable passed to the method
- However, unlike in pass-by-value, the method can modify the actual object that the reference is pointing to, 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={35,45,55};
        //print the array values
        for(int i=0;i<ages.length;i++){
            System.out.println(ages[i]);
        }
        //call the method test() and pass reference to array
        test(ages);

        //print array value again
        for(int i=0;i<ages.length;i++){
            System.out.println(ages[i]);
        }
    }
    public static void test(int[] arr){
        //change the values of array
        for(int i=0;i<arr.length;i++){
            arr[i]=i+40;
        }
    }
}
```

Pass-by-Reference



Defining Your Own Class

Defining classes

- class is defined using the “class” keyword and has the general form as:

```
class classname {  
    type instance-variable1;  
    type instance-variable2;  
    // ...  
    type instance-variableN;  
    type methodname1(parameter-list) {  
        // body of method  
    }  
    type methodname2(parameter-list) {  
        // body of method  
    }  
}
```

Example of a class

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // display volume of a box  
    void volume() {  
        System.out.print("Volume is ");  
        System.out.println(width * height * depth);  
    }  
}
```

Creating an Object

■ The Object for the Box class is created as:


```
Box mybox = new Box();
```

Where

1. mybox is a variable called Object reference
2. new is the operator which allocates memory for the object
3. Box() is default constructor

Constructor is a special method having the same name as the class name and is automatically called when the object is created using “new” operator (more in next slides)

Accessing Instance Variable and Methods

 variables and methods of an object are accessed using the “.”(dot) operator as

```
mybox.width=30; //sets the length to 30  
mybox.height=20; //sets the breadth to 20  
mybox.depth=10; //sets the height to 10  
and  
mybox.volume() ; //calls the method volume()
```

program that uses the **Box** class

```
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox = new Box();  
        // assign values to mybox's instance variables  
        mybox.width = 10;  
        mybox.height = 20;  
        mybox.depth = 15;  
        // call volume() of box  
        mybox.volume()  
    }  
}
```

“class” as user defined data type

- Creating a class also creates a new Data type
- This data type can be used to declare objects of that type
- However, obtaining an object of a class is a two step process

1. declare a variable of the class type

`Box mybox;`

this variable does not define an object rather it refers to an object

2. create the actual physical object using new operator

`mybox = new Box();`

The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.

statement

effect

Box mybox ;

null

mybox

mybox = new Box();



mybox



Object mybox

Assigning Object Reference Variables

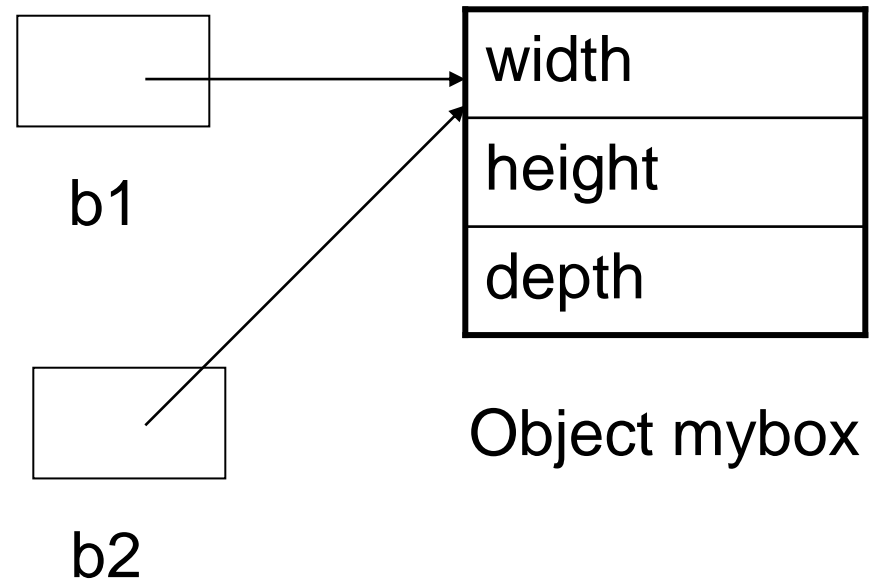
- Object reference variables act differently when an assignment takes place

```
Box b1 = new Box() ;
```

```
Box b2 = b1 ;
```

b1 and **b2** both refer to the same object, they are not linked in any other

When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference.



A Closer look at the Box class

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // display volume of a box  
    void volume() {  
        System.out.print("Volume is ");  
        System.out.println(width * height * depth);  
    }  
}
```

```
class BoxDemo {  
    public static void main(String args[]) {  
        Box mybox = new Box();  
        // assign values to mybox's instance variables  
        mybox.width = 10;  
        mybox.height = 20;  
        mybox.depth = 15;  
        // call volume() of box  
        mybox.volume();  
    }  
}
```

1. The Object oriented design emphasizes on the security of data (instance variables)
2. therefore, instance variables should not be accessed directly using the Object references
3. a better approach to setting the dimensions of a box is to create a method that takes the dimension of a box in its parameters and sets each instance variable appropriately

Adding a Parameterized method initialize the Box

```
// This program uses a parameterized method.
```

```
class Box {  
    double width;  
    double height;  
    double depth;  
    // compute and return volume  
    double volume() {  
        return width * height * depth;  
    }  
  
    // sets dimensions of box  
    void setDim(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
}
```

Using the method to initialize Box

```
class BoxDemo2 {  
    public static void main(String args[]) {  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
        // initialize each box  
        mybox1.setDim(10, 20, 15);  
        mybox2.setDim(3, 6, 9);  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```


Constructors


- It can be tedious to initialize all of the variables in a class each time an instance is created.
- it would be simpler and more concise to have all of the setup done at the time the object is first created
- This automatic initialization is performed through the use of a **constructor**.
- A **constructor** initializes an object immediately upon creation

Constructors

Definition

- A constructor is special method which is automatically called when the instance is created

 A *constructor* initializes an object immediately upon creation

 A *constructor* cannot be called explicitly using the object reference

 A *constructor* has the same name as that of the class

The Default Constructor

- In the absence of any user defined constructor in the class, Java creates a no argument constructor for the class, this constructor is called default Constructor
- The default constructor automatically initializes all instance variables to zero.
- Once a constructor is defined, the **default** constructor is no longer used

Using a Constructor

```
// This program uses a constructor.
class Box {
    double width;
    double height;
    double depth;
// def of constructor
    Box() {
        width = 10
        height =10;
        depth =10;
    }

    // compute and return volume
    double volume() {
        return (width * height * depth);
    }
}
```

Using a Constructor

```
class BoxDemo3 {  
    public static void main(String args[]) {  
        // declare, allocate, and initialize Box objects  
        Box mybox1 = new Box();  
        Box mybox2 = new Box();  
        double vol;  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```

Using a Constructor

 When this program is run, it generates the following results:

Volume is 1000.0

Volume is 1000.0

1. The Box constructor used in the class is not very useful as all the boxes has the same dimension
2. we need a constructor which constructs objects of various dimensions
3. This task can be achieved using a constructor with parameters

Using a Constructor

```
// This program uses a parameterized constructor.
```

```
class Box {  
    double width;  
    double height;  
    double depth;  
  
    // def of constructor  
    Box(double w, double h, double d) {  
        width = w  
        height =h;  
        depth =d;  
    }  
  
    // compute and return volume  
    double volume() {  
        return (width * height * depth);  
    }  
}
```

Using a Constructor

```
class BoxDemo4 {  
    public static void main(String args[]) {  
        // declare, allocate, and initialize Box objects  
        Box mybox1 = new Box(10, 20, 15);  
        Box mybox2 = new Box(3, 6, 9);  
        double vol;  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume is " + vol);  
        // get volume of second box  
        vol = mybox2.volume();  
        System.out.println("Volume is " + vol);  
    }  
}
```


Using a Constructor

Output of the program is:

Volume is 3000.0

Volume is 162.0

“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.

The “**this**” Keyword

// A redundant use of this

```
Box(double w, double h, double d) {  
    this.width = w;  
    this.height = h;  
    this.depth = d;  
}
```

As **this** refers directly to the object, it can be to resolve any name space collisions that might occur between instance variables and local variables.

// Use this to resolve name-space collisions.

```
Box(double width, double height, double depth) {  
    this.width = width;  
    this.height = height;  
    this.depth = depth;  
}
```

Coding Guidelines

- Think of an appropriate name for your class.
Don't just call your class XYZ or any random names you can think of.
- Class names starts with a CAPITAL letter - not a requirement, however.
- The filename of your class must have the SAME NAME as your class name.

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

Overloading Methods

Method overloading

- allows a method with the same name but different parameters, to have different implementations and return values of different types
- can be used when the same operation has different implementations.

Always remember that overloaded methods have the following properties:

- the same method name
- different parameters or different number of parameters
- return types can be different or the same

Overloading methods

```
class Overload {  
    void test() {  
        System.out.println("No parameters");  
    }  
    void test(int a) {  
        System.out.println("a: " + a);  
    }  
    void test(int a, int b) {  
        System.out.println("a and b: " + a + " " + b);  
    }  
    double test(double a) {  
        System.out.println("double a: " + a);  
        return a*a;  
    }  
}
```

Overloading methods

```
class OverloadDemo {  
    public static void main(String args[]) {  
        Overload ob = new Overload();  
        double result;  
        // call all versions of test()  
        ob.test();  
        ob.test(10);  
        ob.test(10, 20);  
        result = ob.test(123.25);  
        System.out.println("Result of ob.test(123.25): " +  
            result);  
    }  
}
```


Overloading methods

The Output

No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

Constructor Overloading

- As Constructors are methods, so method overloading can be applied to constructors
- A class may have more than one overloaded constructors

Constructor Overloading

```
class Box {  
    double width, height, depth;  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    Box(double len) {  
        width = height = depth = len;  
    }  
  
    double volume() {  
        return width * height * depth;  
    }  
}
```

Constructor Overloading

```
class OverloadCons {  
    public static void main(String args[]) {  
  
        // create boxes using the various constructors  
        Box mybox1 = new Box(10, 20, 15);  
        Box mycube = new Box(7);  
        double vol;  
  
        // get volume of first box  
        vol = mybox1.volume();  
        System.out.println("Volume of mybox1 is " + vol);  
  
        // get volume of cube  
        vol = mycube.volume();  
        System.out.println("Volume of mycube is " + vol);  
    }  
}
```

Constructor Overloading

The Output

Volume of mybox1 is 3000.0

Volume of mycube is 343.0

“this()” constructor call

- Constructor calls can be chained, meaning, you can call another constructor from inside another 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.

Constructor Overloading

```
class Box {  
    double width, height, depth;  
    String color;  
    double weight;  
    Box(double w, double h, double d) {  
        width = w;  
        height = h;  
        depth = d;  
    }  
  
    Box(double w, double h, double d, String c) {  
        this(w,h,d);  
        color=c;  
    }  
}
```

//next slide

Constructor Overloading

```
Box(double w, double h, double d, String c, double wt) {  
    this(w,h,d,c);  
    weigth=wt;  
}
```

```
Box(double len) {  
    width = height = depth = len;  
}
```

```
double volume() {  
    return width * height * depth;  
}
```

```
}//end
```

Packages

Packages

Packages

- are Java's means of grouping related classes and interfaces together in a single unit (interfaces will be discussed later).
- This powerful feature provides for a convenient mechanism for managing a large group of classes and interfaces while avoiding potential naming conflicts.

Importing Packages

- To be able to use classes outside of the package you are currently working in, you need to import the package of those classes.
- By default, all your Java programs import the `java.lang.*` package, that is why you can use classes like `String` and `Integer` inside the program even though you haven't imported any packages.
- The syntax for importing packages is as follows:
`import <nameOfPackage>;`

Example: Importing Packages or Class

■ **import** java.io.*;

■ **import** java.net.*;

Placing a Class in a Package

- To place a class in a package, we write the following as the *first line* of the code (except comments)

package <packageName>;

e.g. **package myownpackage;**

- Packages can also be nested. In this case, the Java interpreter expects the directory structure containing the executable classes to match the package hierarchy.

package myowndir.myownsubdir.myownpackage;

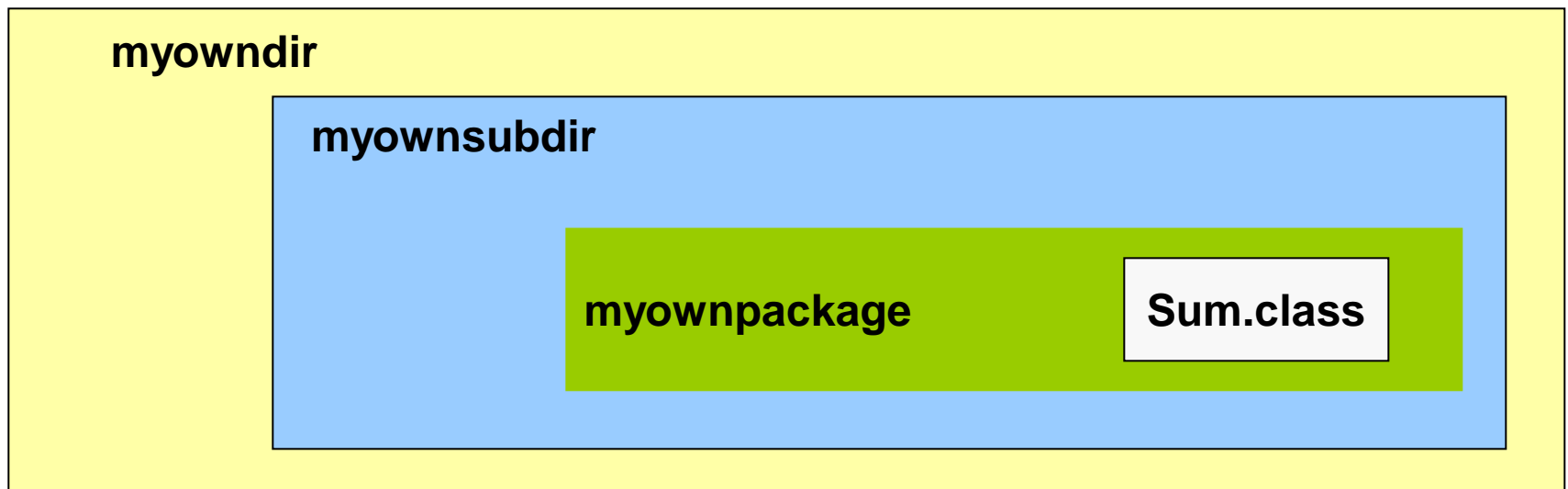
“A package may contain more than one class files

Placing a Class in a Package

■ for example if a class named **Sum** has to be placed under the package

`myowndir.myownsubdir.myownpackage`

we place the class file in the last directory



Creating a package

```
package com.example; //must be the first line of code
```

```
public class Sum
{
    private double a;
    private double b;

    public Sum(double a, double b)
    {
        this.a=a;
        this.b=b;
    }

    public double getSum()
    {
        return (a+b);
    }
}
```

Creating a package

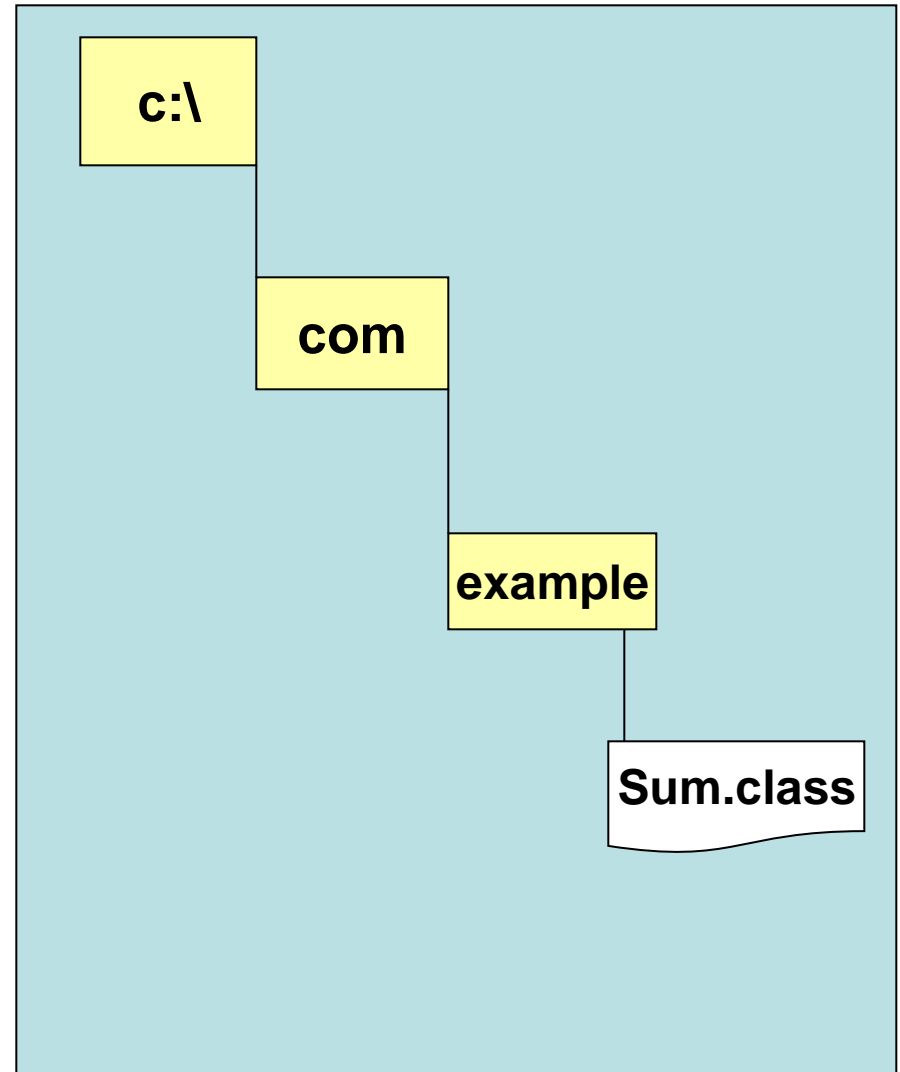
■ Compiling the package

compile the package as:

```
c:\> javac -d dest_path Sum.java
```

-d option tells java compiler to place the class file in the `dest_path`
this option also creates the folder structure to create the package and places the class file in the last folder

e.g. if we compile the file as
`javac -d c:\ Sum.java`
then the result would be as described by the diag.



Using the Package

```
import com.example.Sum;

class UseSum
{
    public static void main(String[] args)
    {
        //create an object of Sum class
        Sum s = new Sum(4,5);

        //display the sum by calling the method getSum()
        System.out.println("The sum = "+s.getSum());
    }
}
```

Note: set the classpath to **c:\com\example** prior to running this example

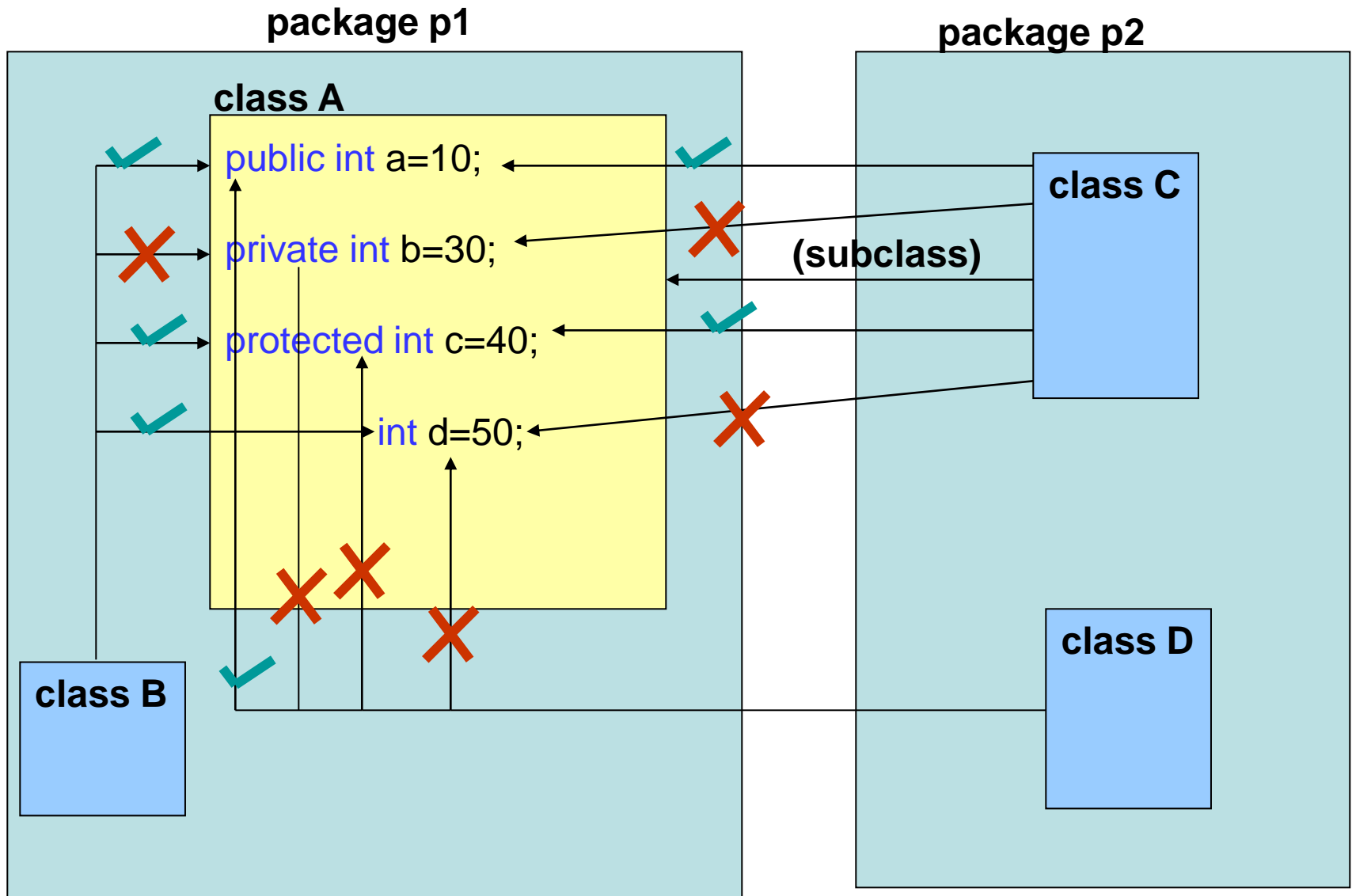
Access Specifiers

Access Modifiers

- There are four different types of member access modifiers in Java:
 1. public
 2. private
 3. protected
 4. **Default Access** (not a keyword here)

- The first three access modifiers are explicitly written in the code to indicate the access type, for the fourth one which is default, no keyword is used.

Access Modifiers at glance



“static” Keyword

- At times we need to define a class member that will be used independently of any object of that class.
- Normally a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- Such members can be defined with a “**prefixed**” keyword “**static**”
- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object.

“static” variables

- `static` variables are declared as:
 `static int i=20;`
 `static String filename="data.txt";`
- Instance variables declared as `static` are, essentially, global variables
- When objects of its class are declared, no copy of a `static` variable is made
- all instances of the class share the same **static** variable

static methods

■ **static** methods are declared as:

static returnType methodName(parameter)

e.g. *static void print(String name);*

■ Methods declared as **static** have several restrictions:

1. They can only call other **static** methods.
2. They must only access **static** data.
3. They cannot refer to **this** or **super** in any way. (The keyword **super** relates to inheritance)

static Blocks

- If some computation is required in order to initialize **static** variables, a **static** block can be declared
- a static block gets executed exactly once, when the class is first loaded.
- Syntax for static block:

```
static {  
    // the code to be executed  
}
```


Accessing **static** members

- The static members gets automatically initialized when JVM loads the class for the first time
- They can be accessed without creating an object of the class
- Accessing a static variable:
classname.variablename

Accessing a static method:
classname.methodname

```
class UseStatic {  
    static int a = 3;  
    static int b;  
    static void meth(int x) {  
        System.out.println("x = " + x);  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
    }  
    static {  
        System.out.println("Static block initialized.");  
        b = a * 4;  
    }  
};
```

```
.....  
class UseStaticDemo{  
    public static void main(String args[]) {  
        UseStatic.meth(42);  
    }  
}
```