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.

Java Background: History

- Java
  - The original motivation for Java
    - The need for platform independent language that could be embedded in various consumer electronic products like toasters and refrigerators.
  - One of the first projects developed using Java
    - a personal hand-held remote control named Star 7.
  - At about the same time, the World Wide Web and the Internet were gaining popularity. Gosling et. al. realized that Java could be used for Internet programming.

Java Background :What is Java Technology?

The Java technology is:

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

Java Technology: Programming Language

- As a programming language, Java can create all kinds of applications that you could create using any conventional programming language.

Java Technology:
A Development Environment

- Java development environment, provides you with a large Collection of tools:
  - A compiler (javac)
  - An interpreter (java)
  - A documentation generator (javadoc)
  - A class file packaging tool and so on...

Java Technology: An Application and Runtime Environment

- Java technology applications are typically general-purpose programs that run on any machine where the Java runtime environment (JRE) is installed.

- There are two main deployment environments:

1. The JRE supplied by the Java 2 Software Development Kit (SDK) contains the complete set of class files for all the Java technology packages, which includes basic language classes, GUI component classes, and so on.

1. The other main deployment environment is on your web browser. Most commercial browsers supply a Java technology interpreter and runtime 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

Java Features : 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

Java Features : 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

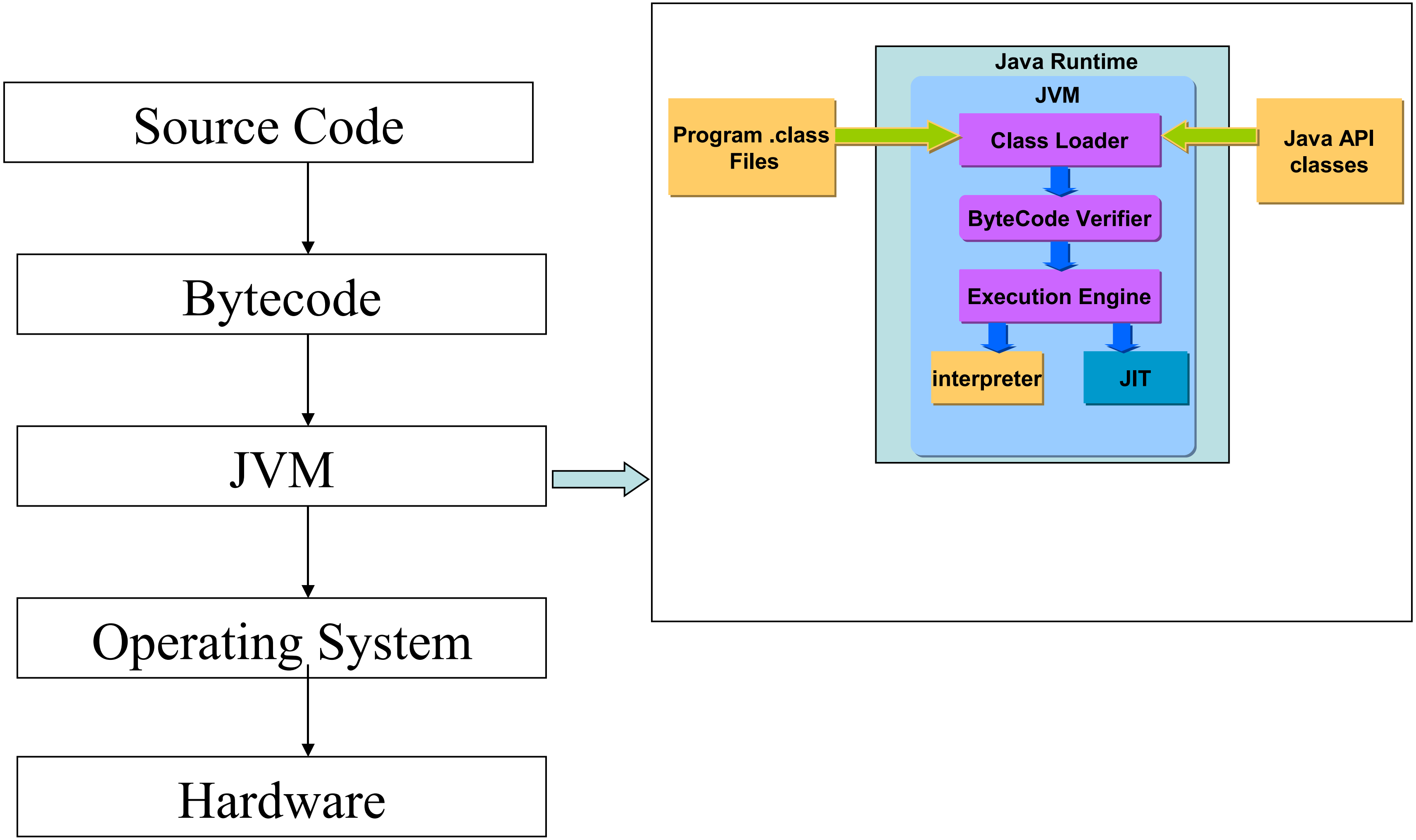
Java Features : Code Security

- **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

Java Features : 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

The Java Architecture



Source Code

Bytecode

JVM

Operating System

Hardware

Java Runtime

JVM

Program .class Files

Class Loader

Java API classes

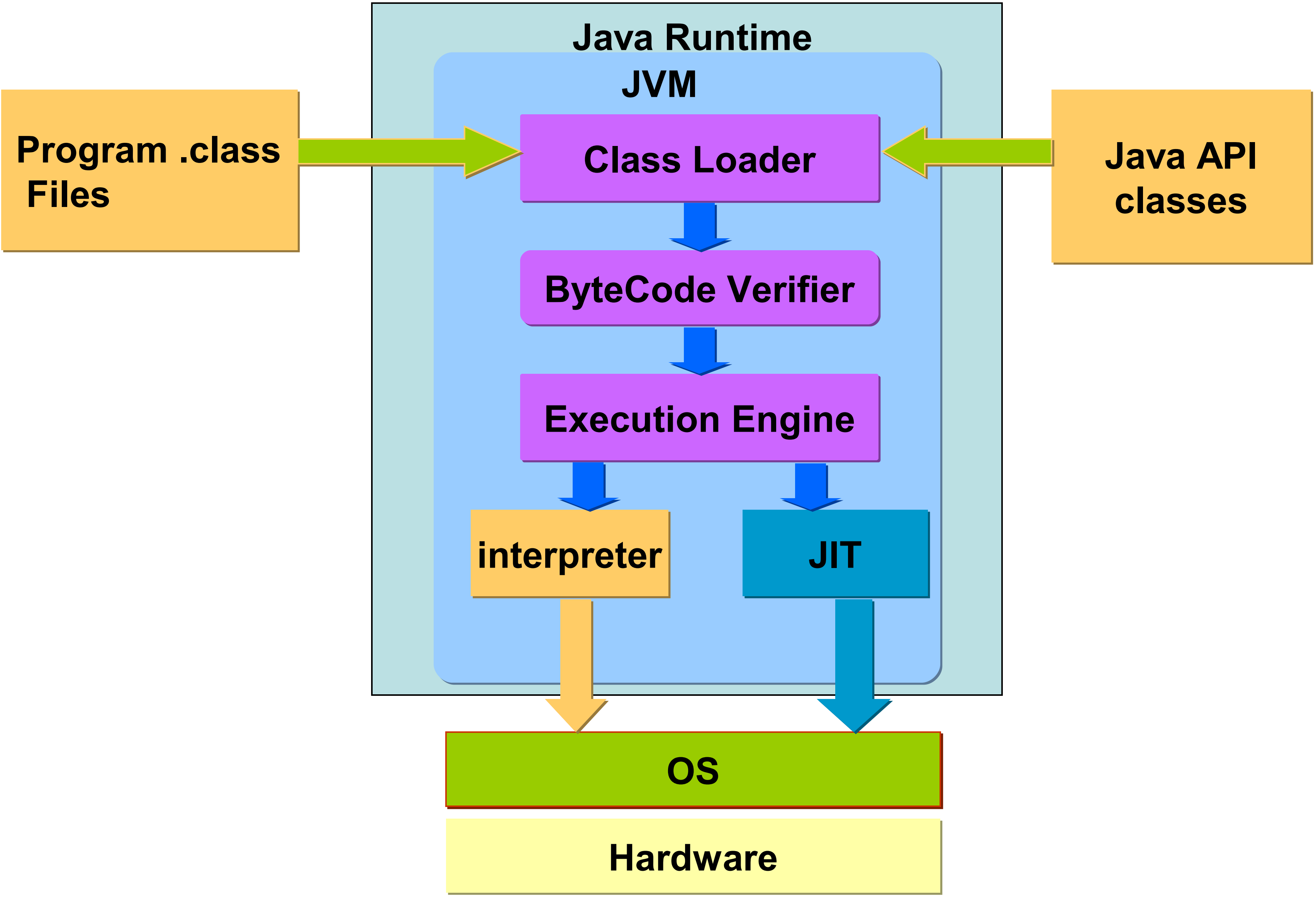
ByteCode Verifier

Execution Engine

interpreter

JIT

Java Virtual Machine



The Java Buzzwords

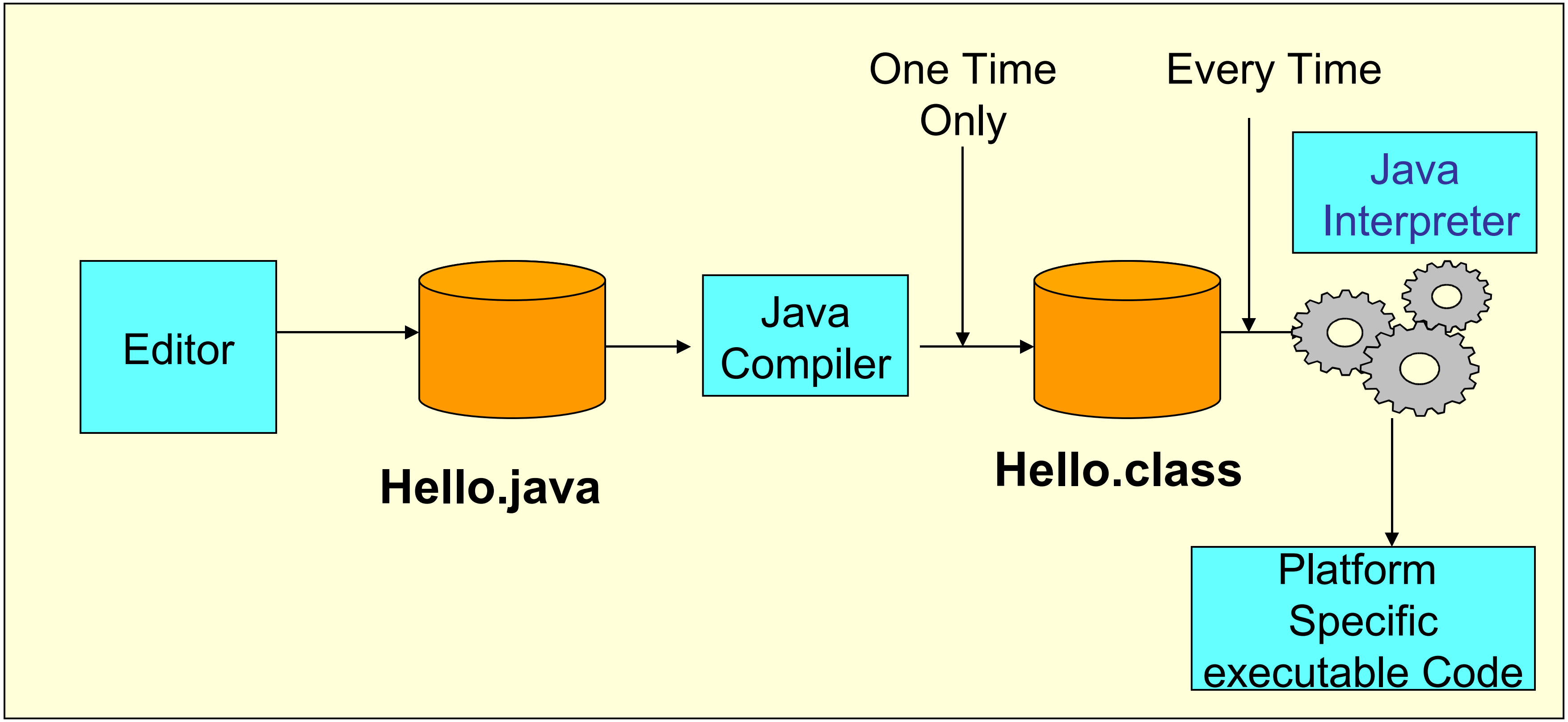
- Simple
  - Small language [large libraries]
  - Small interpreter (40 k), but large runtime libraries(175 k)
- Object-Oriented
  - Supports encapsulation, inheritance, abstraction, and polymorphism.
- Distributed
  - Libraries for network programming
  - Remote Method Invocation
- Architecture neutral
  - 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

Getting to know your Programming Environment

Definitions

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

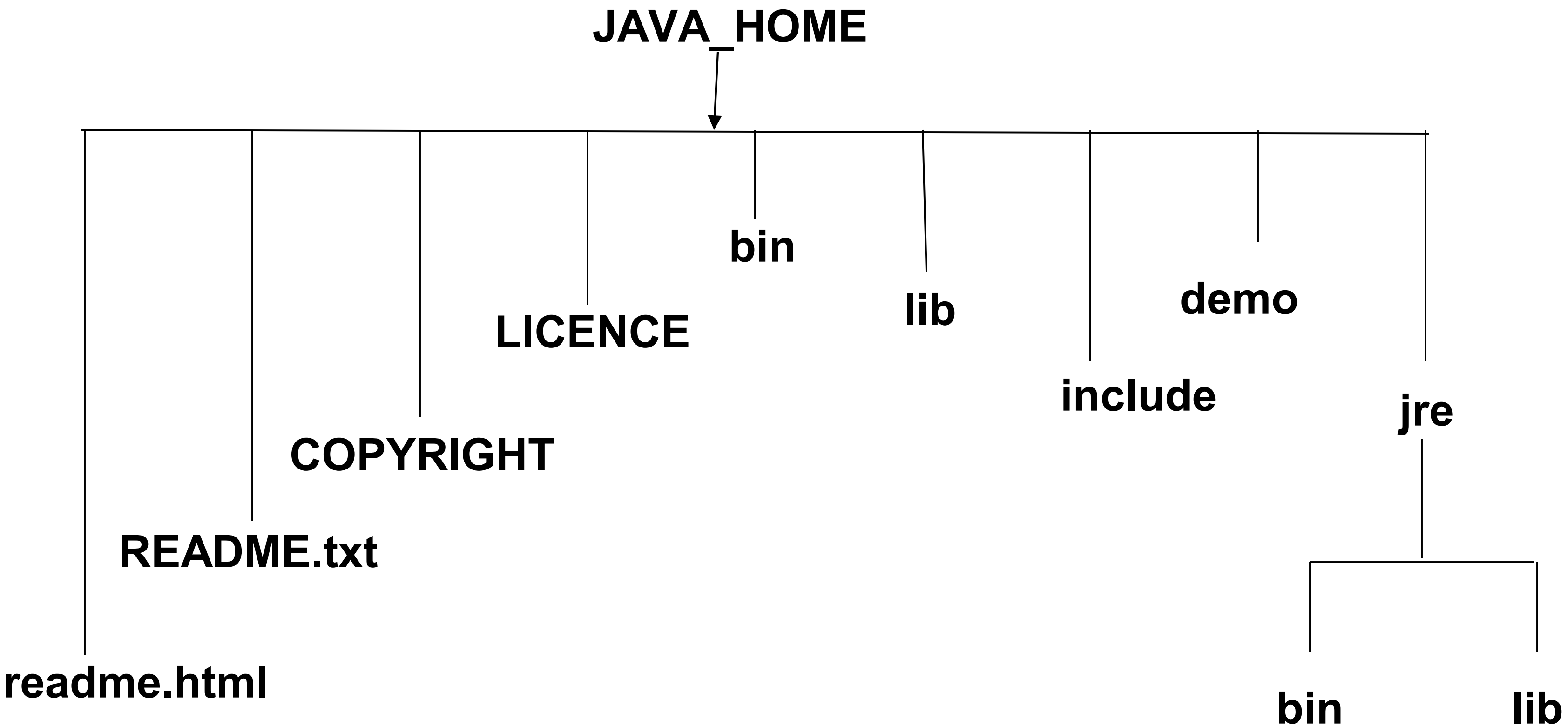
Definitions

- **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



Installation and Setup JDK

- **set the JAVA_HOME variable**
  - JAVA_HOME is the absolute address of the java installation directory
- set JAVA_HOME for Windows NT/2000/XP
  - Choose Start→ Settings→ Control Panel→double-click System
  - Select the Advanced tab → the Environment tab
  - click new type
    - variable name → JAVA_HOME
    - variable value → absolute address of java installation folder **e.g j2sdk1.4.2**
- JAVA_HOME variable will be used later to set PATH and CLASSPATH environment variables

Installation and Setup JDK

- **Update the PATH variable**
  - To set the PATH permanently, add the full path of JAVA_HOME**bin** directory to the PATH variable
- Update PATH for Windows NT/2000/XP
  - Choose Start→ Settings→ Control Panel→double-click System
  - Select the Advanced tab → the Environment tab
  - look for PATH variable and append

JAVA_HOME\bin

- e.g. j2sdk1.4.2\bin

  - The new path takes effect in each new Command Prompt window you open after setting the PATH variable

Installation and Setup JDK

- **Update the PATH variable** (2)
  - To set the PATH using the Command Prompt add the full path of JAVA_HOME**bin** directory to the PATH variable
- Update PATH for Windows NT/2000/XP
  - Open the command prompt
    - select Start→run and type cmd then hit enter key
  - to set the PATH Variable at the Prompt type

set PATH=JAVA_HOME\bin;%path%

- e.g. set path=c:\j2sdk1.4.2\bin;%path%
- The new path takes effect immediately and is valid only for the current Command Prompt window

Installation and Setup JDK

- **Update the CLASSPATH variable**
  - The **classpath** is a user defined environment variable used by Java to determine where predefined classes are located.
- **set CLASSPATH for Windows NT/2000/XP**
  - Choose Start→ Settings→ Control Panel →double-click System
  - Select the Advanced tab → the Environment variables tab
  - click new and type
    - variable name → CLASSPATH
    - variable value → classpath for the required classes/jar files
  - OR look for CLASSPATH variable and append the new **classpath**
    - **e.g c:\j2sdk1.4.2\lib\tools.jar**
    - **e.g. c:\j2sdk1.4.2\jre\lib\rt.jar**
  - classpath variable also can be set using the command prompt as in the case of setting PATH

Installation and Setup JDK

- **Update the PATH variable** (2)
  - To set the PATH using the Command Prompt add the full path of JAVA_HOME**bin** directory to the PATH variable
- Update PATH for Windows NT/2000/XP
  - Open the command prompt
    - select Start→run and type cmd then hit enter key
  - to set the PATH Variable at the Prompt type

set PATH=JAVA_HOME\bin;%path%

- e.g. set path=c:\j2sdk1.4.2\bin;%path%

- The new path takes effect immediately and is valid only for the current Command Prompt window

- The classpath is a user defined environment variable used by Java to determine where predefined classes are located.
- These instructions show how to create it and set its value

My First Java Program

```
public class Hello {

        /**
        * My first Java program
         */
public static void main( String[] args ){

                //prints the string “Hello world” on screen
                System.out.println(“Hello world”);
         }
}
```

Using Text Editor and Console

- Step 1: Start the Text Editor
  - To start the Text Editor in Windows,

    click on Start->Programs -> Accessories-> Notepad
- Step 2: Open Command Prompt
  - To open Command Prompt in Windows, click on Start->Programs -> Accessories -> Command Prompt
- Step 3: Write your the source code of your Java program in the Text Editor

Using Text Editor and Console

- **Step 4: Save your Java Program**
  - Filename: Hello.java
  - Folder name: MYJAVAPROGRAMS
  - To open the Save dialog box, click on the File menu found on the menubar and then click on Save.
  - If the folder MYJAVAPROGRAMS does not exist yet, create the folder

Using Text Editor and Console

- **Step 5: Compiling your program**
  - Go to the Command Prompt window
  - Go to the folder MYJAVAPROGRAMS where you saved the program
  - To compile a Java program, we type in the command: javac [filename]
  - So in this case, type in:

```
javac Hello.java
```

- **During compilation, javac adds a file to the disk called [filename].class, or in this case, `Hello.class`, which is the actual bytecode.**

Using Text Editor and Console

- **Step 6: Running the Program**
  - To run your Java program, type in the command:

    java [filename without the extension]
  - so in the case of our example, type in:
  - **java Hello**

- You can see on the screen after running the program:

  "Hello world!"

Programming Fundamentals

Introduction to OO Programming

Classes and Objects

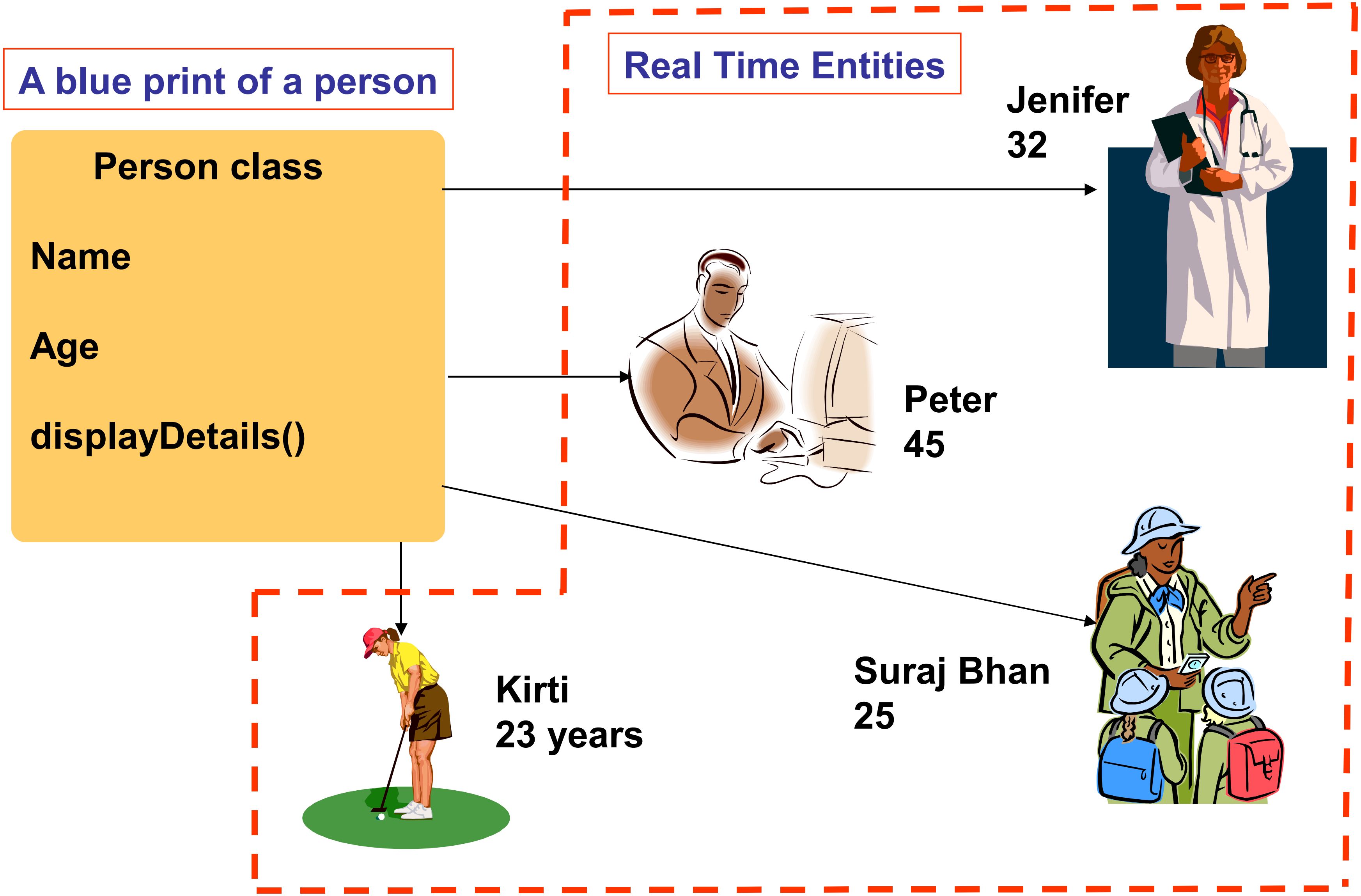
- **Class**
  - can be thought of as a template, a prototype or a blueprint of an object
  - A class is a structure that defines the data and the methods to work on that data.
  - is the fundamental structure in object-oriented programming
  - is a user defined data type

- **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.

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”

Dissecting my First Java Program

```
public class Hello
 {
          /**
           * My first Java program
           */
         public static void main( String[] args ){

         //prints the string Hello world on screen
            System.out.println(“Hello world”);

             }
  }
```

Coding Guidelines

- Your Java programs should always end with the **.java** extension.

- Filenames should match the name of your public class. So for example, if the name of your public class is Hello, you should save it in a file called **Hello.java**.

- You should write comments in your code explaining what a certain class does, or what a certain method do.

Java Comments

- Comments
  - These are notes written to a code for documentation purposes.
  - Those texts are not part of the program and does not affect the flow of the program.
- 3 Types of comments in Java
  - C++ Style Comments
  - C Style Comments
  - Special Javadoc Comments

Java Comments

- C++-Style Comments

  // This is a C++ style or single line comments

- C Style comments

  /* this is an example of a C style or multiline comments */

Java Comments

- Special Javadoc Comments
  - Special Javadoc comments are used for generating an HTML documentation for your Java programs.
  - You can create javadoc comments by starting the line with /** and ending it with */.
  - Like C-style comments, it can also span lines.
  - It can also contain certain tags to add more information to your comments.
- For example:

```
/** This is an example of special java doc
comments used for \n generating an html
documentation. It uses tags like:
@author Florence Balagtas
@version 1.2
*/
```

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

🟥 **reserved words but not used in java**

Data Types in Java

- The Java programming language has two categories of data types

1. primitive
2. reference

Primitive Data Types

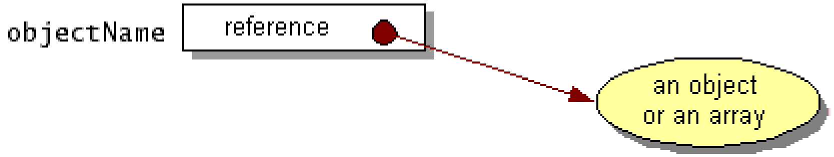
Keyword	Description/Size (bytes)	Example
	Integer Type	
byte	1	
short	2	
int	4	178
long	8	**8864L**
	Real numbers	
float	4	87.363F
double	8	**37.266** or 37.266D
	Other Types	
char	2	**'c'**
boolean	true/false	
void	nothing	

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.



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

```
public class OutputVariable {
public static void main( String[] args ){
           int value = 10;
          char x;
           x = ‘A’;

          System.out.println( value );
           System.out.println( “The value of x=“ + x );
           }
}
```

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

Reference Variables vs. Primitive Variables

- Two types of variables in Java:
  - Primitive Variables
  - Reference 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.

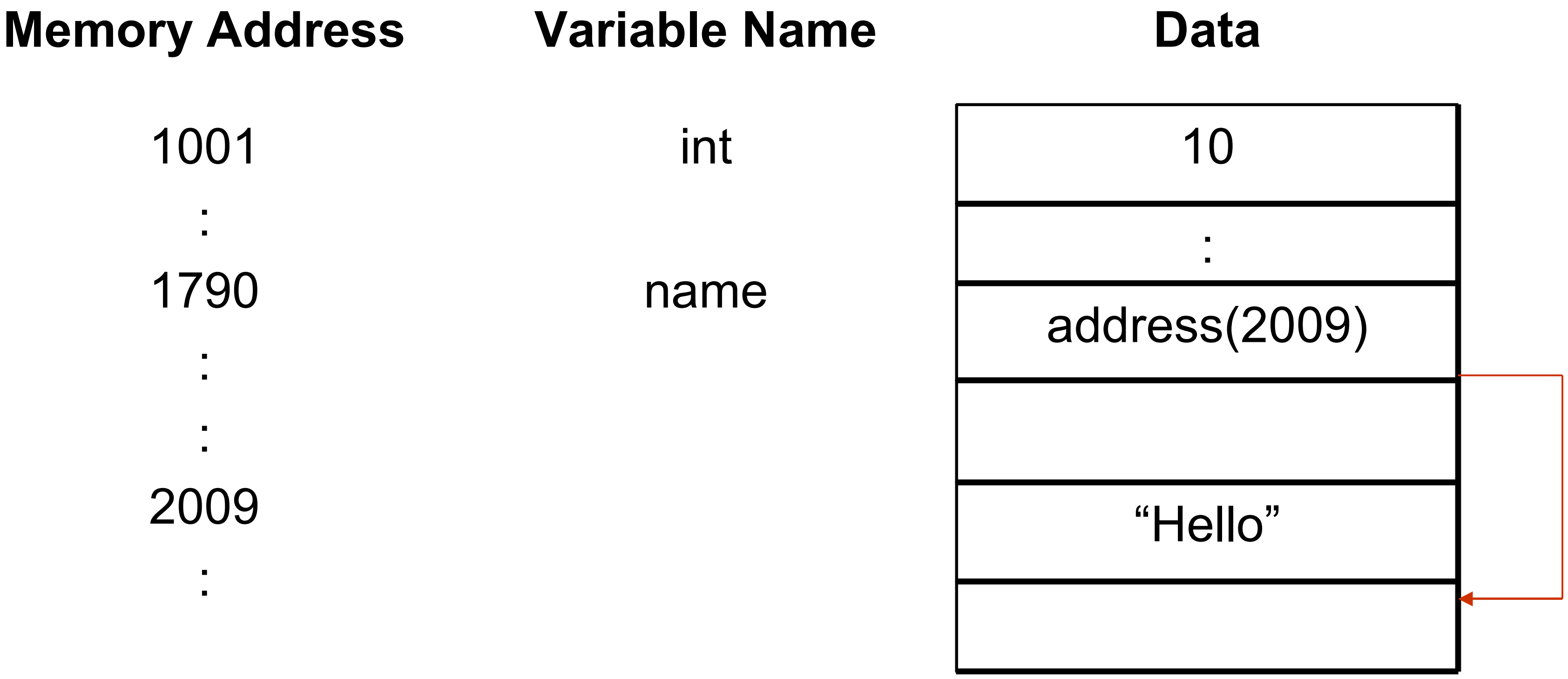
Example

- 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”
:		

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
+	op1 + op2	Adds op1 and op2
-	op1 - op2	Subtracts op2 from op1
*	op1 * op2	Multiplies op1 by op2
/	op1 / op2	Divides op1 by op2
%	op1 % op2	Computes the remainder of dividing op1 by op2

Increment and Decrement Operators

- unary increment operator (++)
- unary decrement operator (--)
- Increment and decrement operators increase and decrease a value stored in a number variable by 1.
- For example, the expression,

```
count=count + 1;//increment the value of
  count by 1
```

is equivalent to,

```
  count++;
```

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, op1 `op` op2

- where, op1, op2 - can be boolean expressions, variables or constants

- op - is either &&, &, ||, | or ^ 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
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

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
(*params*)	Delimits a comma-separated list of parameters
(*type*)	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

Operator	Use	Description
?:	op1 ? op2 : op3	If op1 is true, returns op2. Otherwise, returns op3.
[]	*type* []	Declares an array of unknown length, which contains *type* elements.
[]	*type*[op1]	Creates and array with op1 elements. Must be used with the new operator.
[]	op1[op2]	Accesses the element at op2 index within the array op1. Indices begin at 0 and extend through the length of the array minus one.
.	op1.op2	Is a reference to the op2 member of op1.
()	op1(*params*)	Declares or calls the method named op1 with the specified parameters. The list of parameters can be an empty list. The list is comma-separated.
(type)	(type) op1	Casts (converts) op1 to type. An exception will be thrown if the type of op1 is incompatible with type.
new	new op1	Creates a new object or array. op1 is either a call to a constructor, or an array specification.
instanceof	op1 instanceof op2	Returns true if op1 is an instance of op2

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.`

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	Accelarate		
	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

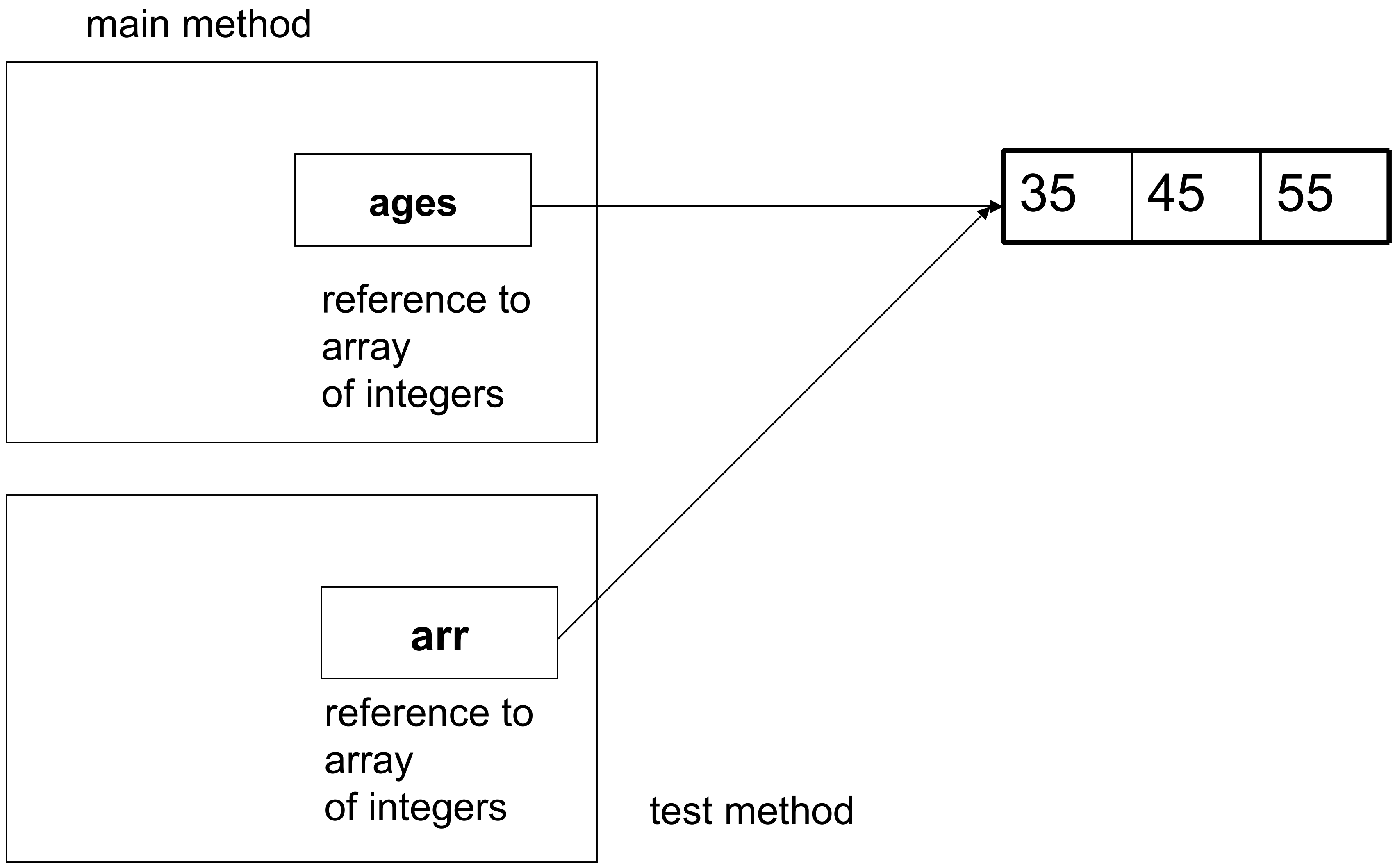
- **Pass-by-Reference**
  - When a pass-by-reference occurs, the reference to an object is passed to the calling method. This means that, the method makes a copy of the reference of the variable passed to the method
  - However, unlike in pass-by-value, the method can modify the actual object that the 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



main method

ages

reference to
array
of integers

arr

reference to
array
of integers

test method

35	45	55

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
1. new is the operator which allocates memory for the object
1. 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

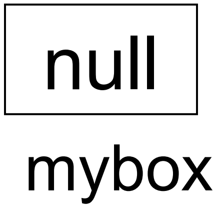
1. create the actual physical object using new operator

   mybox = new Box();

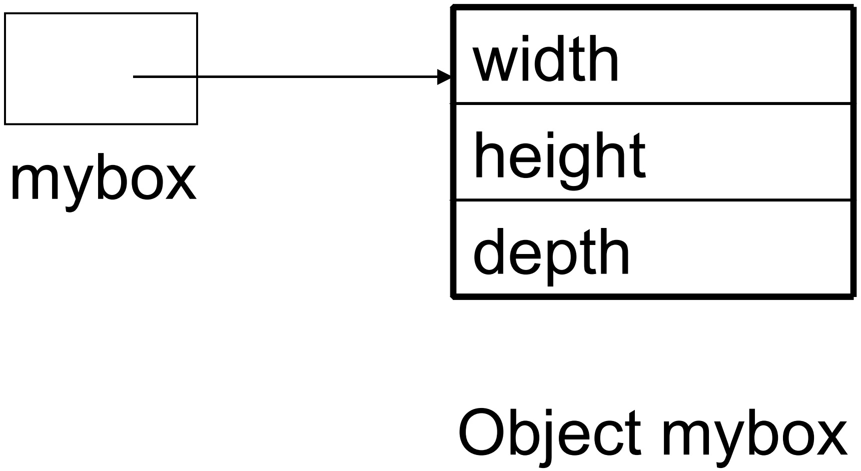
   The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.

statement effect

Box mybox ;



mybox = new Box();

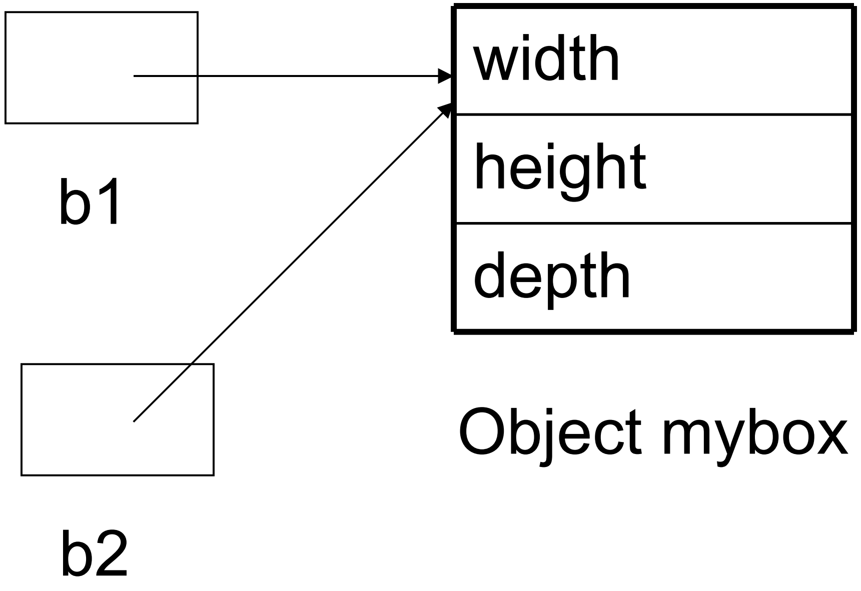


Assigning Object Reference Variables

- Object reference variables act differently when an assignment takes place

```
Box  b1 = new Box() ;

 Box  b2 = b1 ;
```



b1

b2

width

height

depth

Object mybox

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

1. we need a constructor which constructs objects of various dimensions

1. 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.
- Encaptulation

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 Integers 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;

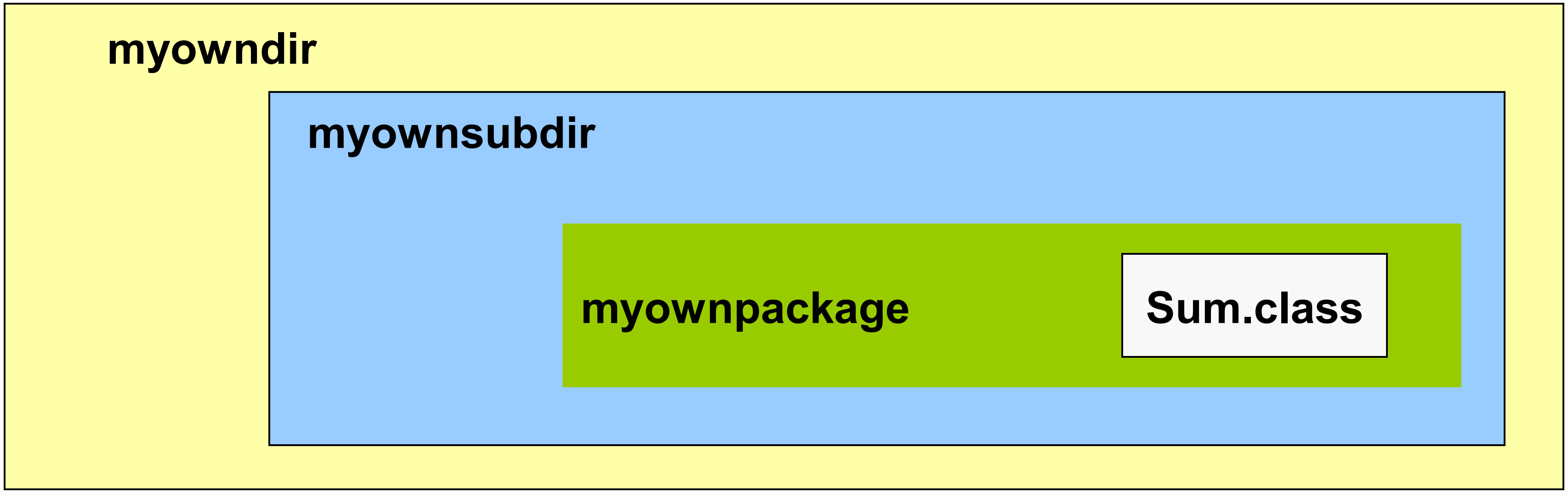
"A package may contain more than one class files

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

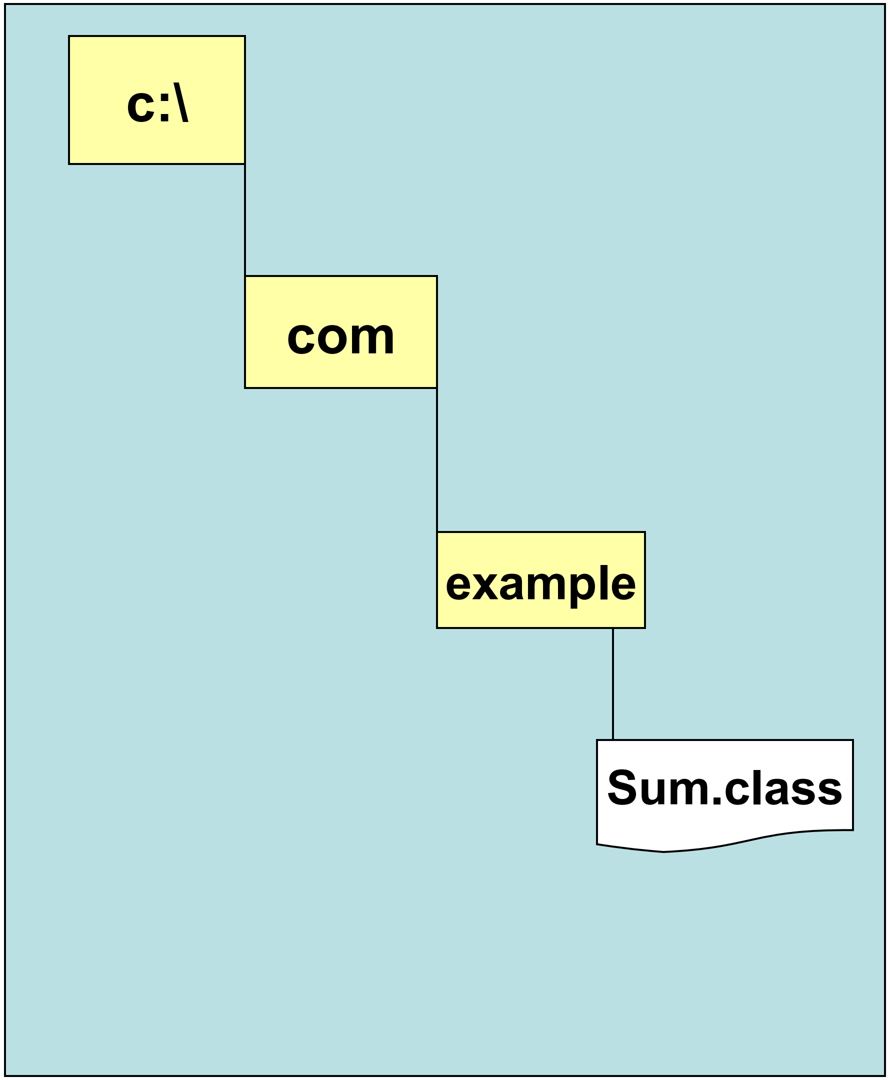
this option also creates the folder structure to create the package and places the class file in the last folder

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 Modifiers

Access Modifiers

- There are four different types of member access modifiers in Java:
  1. public
  2. private
  3. protected
  4. Default

- 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

- Default access
  - specifies that only classes in the same package can have access to the class' variables and methods
  - no actual keyword for the default modifier; it is applied in the absence of an access modifier.

Access Modifiers

Example

```
public class StudentRecord {
    //default access to instance variable
            int name;
    //default access to method
            String getName(){
                                        return name;
                        }
    }
```

Access Modifiers

- **public access**
  - specifies that class members (variables or methods) are accessible to anyone, both inside and outside the class and outside of the package.
  - Any object that interacts with the class can have access to the public members of the class.
  - Keyword: **public**

Access Modifiers

- **Example: “public” Access Modifer**

```
public class StudentRecord {
   //default access to instance variable
     public int name;
     //default access to method
     public String getName(){
                                   return name;
               }
}
```

Access Modifiers

- protected access
  - specifies that the class members are accessible only to methods in that class and the subclasses of the class.
  - Keyword: **protected**

Access Modifiers

- **Example: “protected” Access Modifier**

```
public class StudentRecord {
   //default access to instance variable
      protected int name;
               //default access to method
               protected String getName(){
                           return name;
                     }
}
```

Access Modifiers

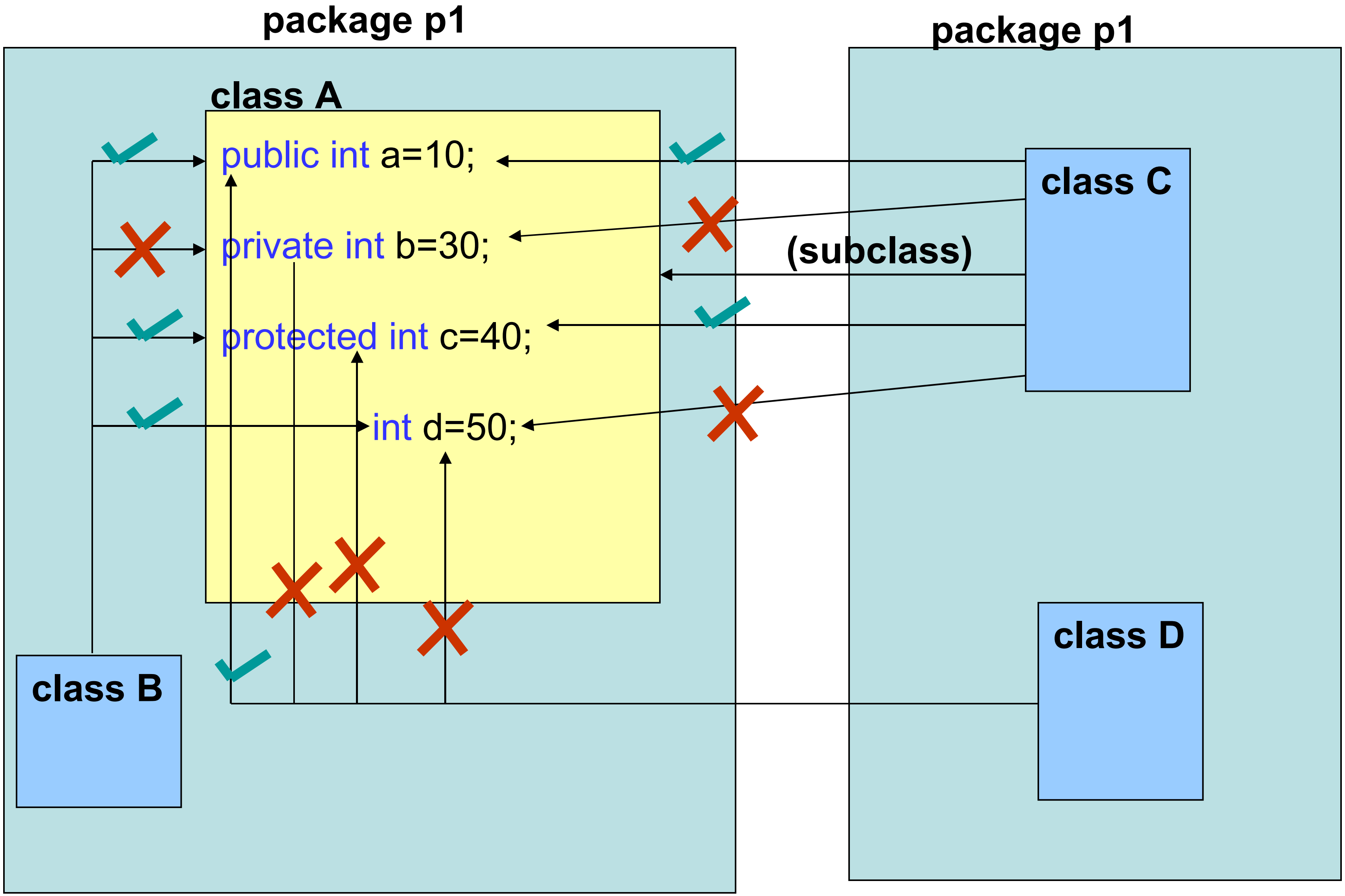
- **private access**
  - specifies that the class members are only accessible by the class they are defined in.
  - Keyword: **private**

Access Modifiers

- **Example: “private” Access Modifier**

```
public class StudentRecord {
    //default access to instance variable
    private int name;
    //default access to method
    private String getName(){
                return name;
          }
}
```

Access Modifiers at glance



Coding Guidelines

- The instance variables of a class should normally be declared private, and the class will just provide accessor and mutator methods to these variables.

“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);
    }
}
```

Type Casting

Type Casting

- Type Casting
  - Converting data type of one data to another data type
- 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:

  (dataType)value

- where, dataType - is the name of the data 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;
```

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.

Converting Primitive types to Objects and vice versa

- 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 instead of int, Double instead of double, and the like)
  - Using the classes that correspond to each primitive type, you can create an object that holds the same value.

Converting Primitive types to Objects 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.

Classes and Objects Again

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 (where Class is itself a class) that has a method called getName().

- In turn, 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

- Classes and Objects
  - Instance variables
  - Class Variables
- Class Instantiation
- Methods
  - Instance methods
  - Passing Variables in Methods (Pass-by-value, Pass-byreference)
  - 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