CODERINDEED

CSE310: Programming in Java

Topic: Java Platform Overview

Outlines

[Expected Time: 3 Hours]

- Introduction
- Why Java?
- Where it is used?
- Characteristics of Java
- Java Platform Editions
- Evolution of Java
- Understanding JDK, JRE and JVM
- How Java is platform-independent
- Evaluating Java libraries, middle-ware, and database options

Introduction

- Java is a programming language and a platform.
- Java is a high level, robust, secured and object-oriented programming language.
- Developed by Sun Microsystems (James Gosling).

Platform: *Any hardware or software environment in which a program runs, is known as a platform.*

Why Java?

- Java is the Internet programming language.
- Java enables users to develop and deploy applications on the Internet for servers, desktop computers, and small hand-held devices.

Where it is Used?

- According to Sun, 3 billion devices run java.

- Desktop Applications such as acrobat reader, media player, antivirus etc.
- Web Applications such as irctc.co.in, javatpoint.com etc.
- Enterprise Applications such as banking applications.
- Mobile Applications
- Embedded System, Smart Card, Robotics, Games etc.

Characteristics of Java

- Java Is Simple
- Java Is Object-Oriented
- Java Is Distributed
- Java Is Interpreted
- Java Is Robust
- **Java Is Secure**
- **Java Is Architecture-Neutral**
- Java Is Portable
- Java's Performance
- Java Is Multithreaded
- Java Is Dynamic

Java Platform Editions

- A Java Platform is the set of APIs, class libraries, and other programs used in developing Java programs for specific applications.

There are 3 Java Platform Editions

1. Java 2 Platform, Standard Edition (J2SE)
   - Core Java Platform targeting applications running on workstations

2. Java 2 Platform, Enterprise Edition (J2EE)
   - Component-based approach to developing distributed, multi-tier enterprise applications

3. Java 2 Platform, Micro Edition (J2ME)
   - Targeted at small, stand-alone or connectable consumer and embedded devices

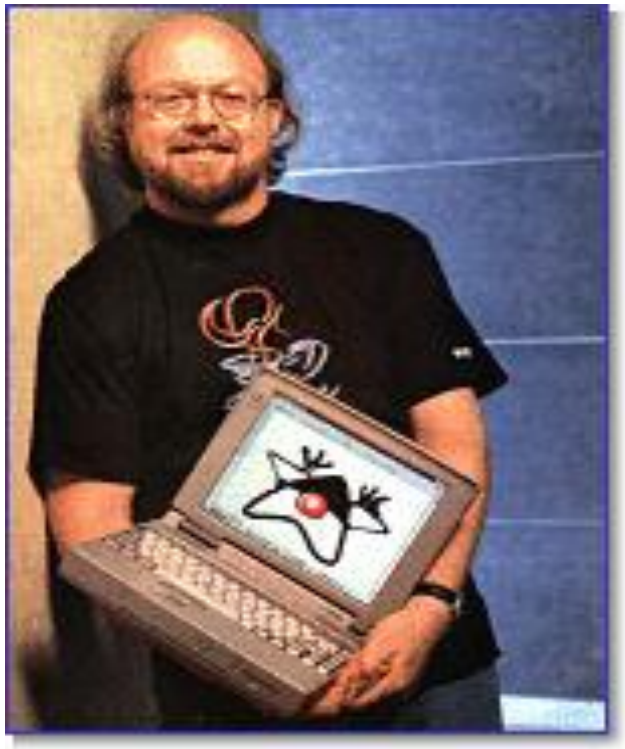
Origin of Java

James Gosling & Patrick Naughton at 1990

Goal : to develop distributed system which is applicable to electronic products(platform independent)

James Gosling

- James Gosling is generally credited as the inventor of the Java programming language



- He was the first designer of Java and implemented its original compiler and virtual machine
- He is also known as the Father of Java.

Brief History of Java

- In 1990, Sun Microsystems began an internal project known as the ***Green Project*** to work on a new technology.

- In 1992, the Green Project was spun off and its interest directed toward building highly interactive devices for the cable TV industry. This failed to materialize.

- In 1994, the focus of the original team was re-targeted, this time to the use of Internet technology. A small web browser called ***HotJava*** was written.
- Oak was renamed to ***Java*** after learning that Oak had already been trademarked.

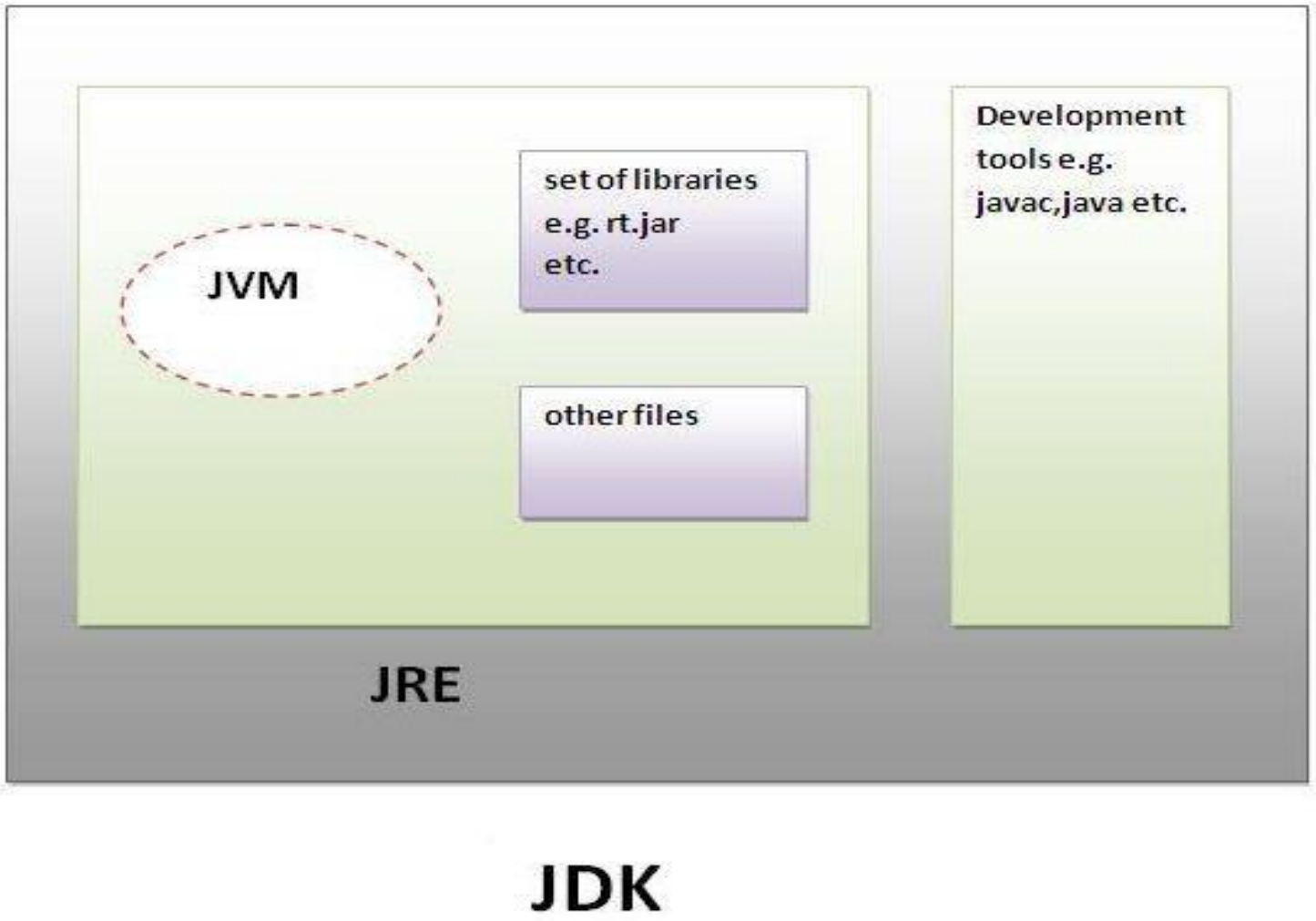
- In 1995, Java was first publicly released.
- In 1996, Java Development Kit (***JDK***) 1.0 was released.
- In 2002, JDK 1.4 (codename *Merlin*) was released, the most widely used version.
- In 2004, JDK 5.0 (codename *Tiger*) was released.

JDK Versions

CODERINDEED

- JDK Alpha and Beta (1995)
- JDK 1.0 (23rd Jan, 1996)
- JDK 1.1 (19th Feb, 1997)
- J2SE 1.2 (8th Dec, 1998)
- J2SE 1.3 (8th May, 2000)
- J2SE 1.4 (6th Feb, 2002)
- J2SE 5.0 (30th Sep, 2004)
- Java SE 6 (11th Dec, 2006)
- Java SE 7 (28th July, 2011)
- Java SE 8 (18th March, 2014)

Understanding JDK, JRE and JVM



Understanding JDK & JRE

JDK

- JDK is an acronym for *Java Development Kit*.
- It physically exists. It contains JRE and development tools.

JRE

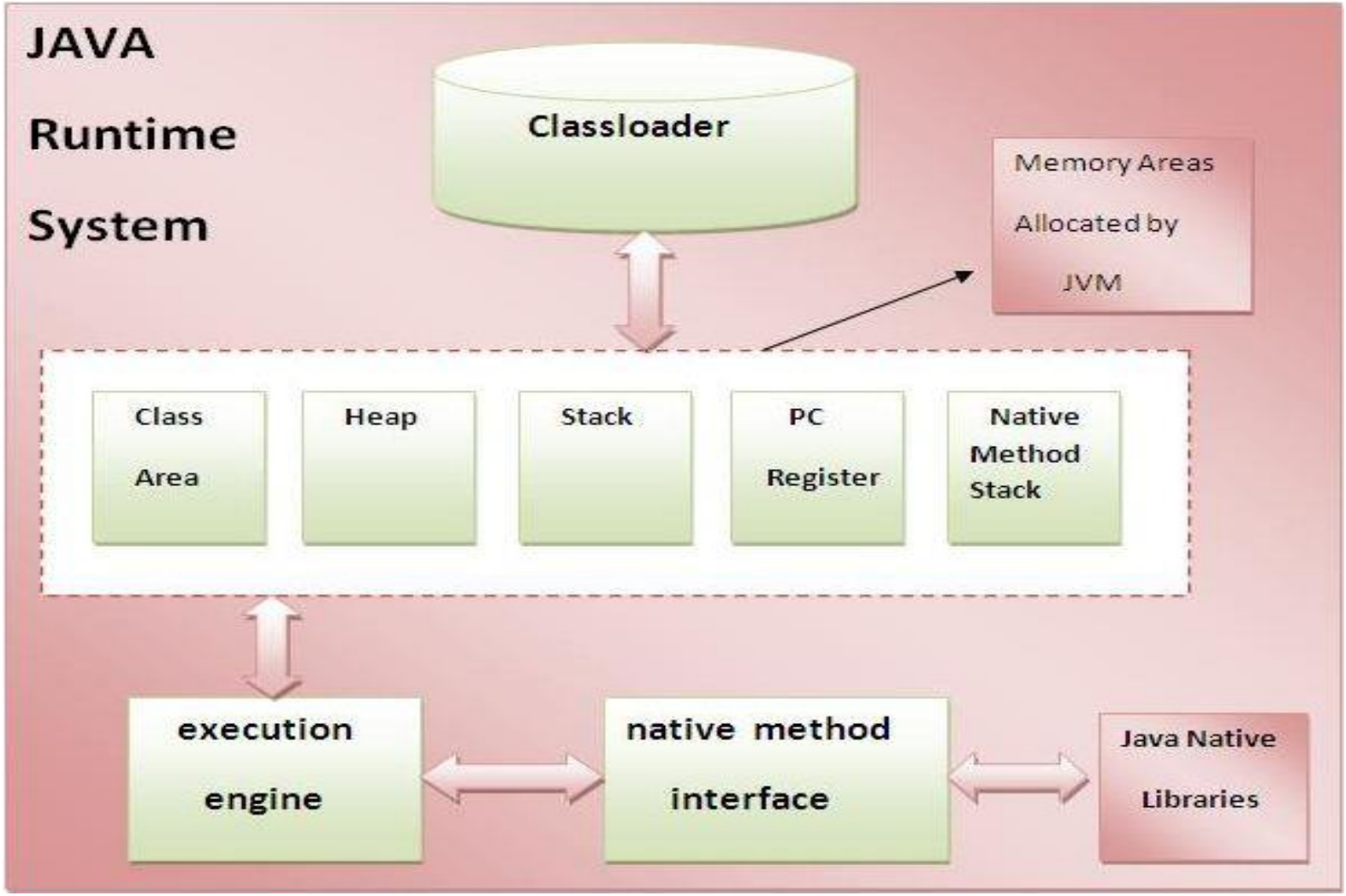
- JRE is an acronym for *Java Runtime Environment*.
- It is the implementation of JVM and used to provide runtime environment.
- It contains set of libraries and other files that JVM uses at runtime.

Understanding JVM

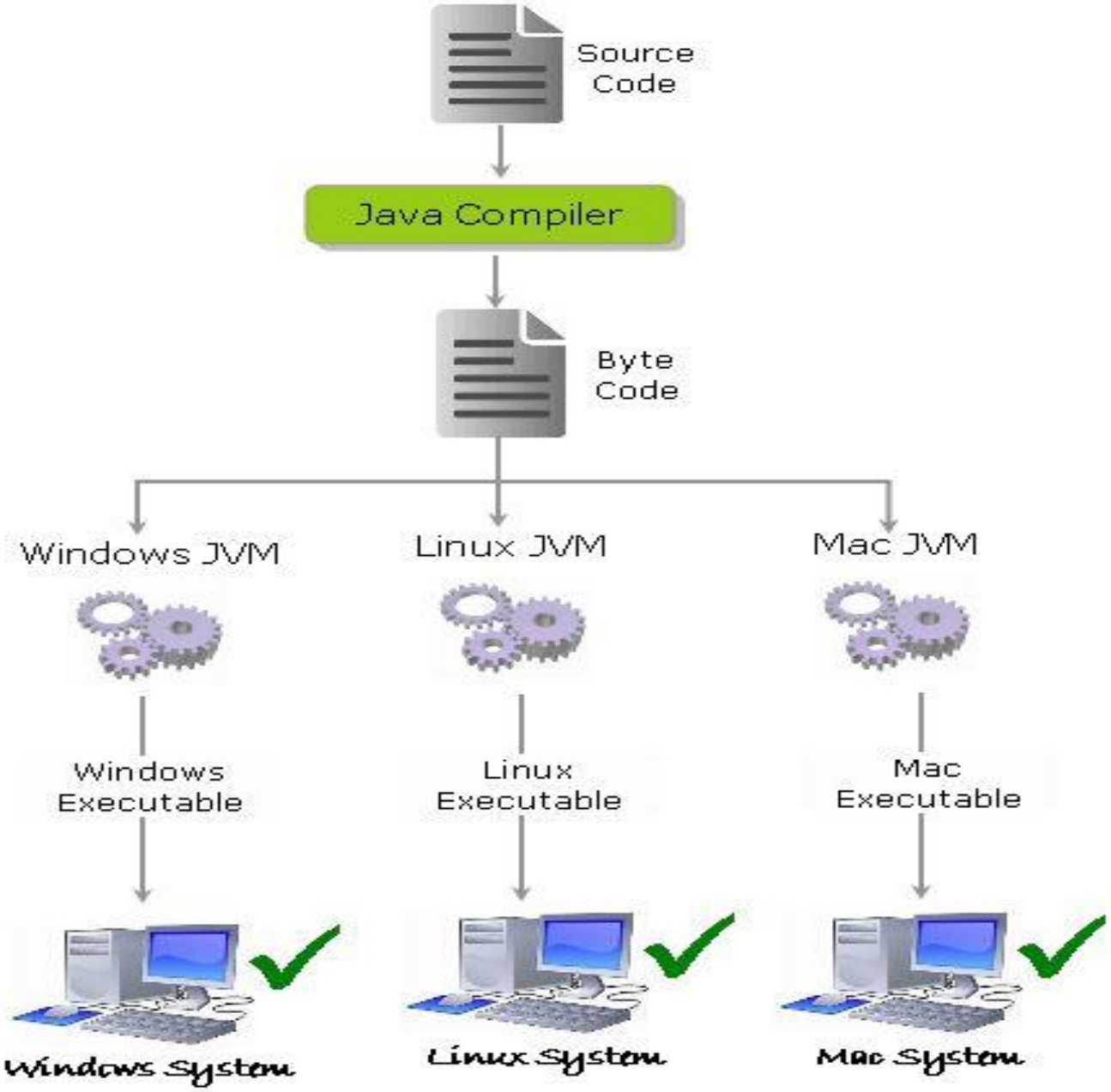
- JVM (Java Virtual Machine) is an abstract machine.
- It is a specification that provides runtime environment in which java byte code can be executed.
- JVMs are available for many hardware and software platforms.

- The JVM performs following main tasks:
  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment

Internal Architecture of JVM



How Java is Platform-independent?



coderindeed.in

How Java is Platform-independent?

- The source code (program) written in java is saved as a file with .java extension.

- The java compiler "javac" compiles the source code and produces the platform independent intermediate code called BYTE CODE. It is a highly optimized set of instructions designed to be executed by the JVM.

How Java is Platform-independent?

- The byte code is not native to any platform because java compiler doesn't interact with the local platform while generating byte code.

- It means that the Byte code generated on Windows is same as the byte code generated on Linux for the same java code.

- The Byte code generated by the compiler would be saved as a file with .class extension. As it is not generated for any platform, can't be directly executed on any CPU.

Portability Vs Platform Independence

- Portability focuses on adaptation of software in various OS, by recompiling the source to make the binary compatible with the target OS and not necessarily modifying the source.

- Platform independence focuses on ability of software to run on VIRTUAL hardware that in turn interfaces with the PHYSICAL hardware.

- Examples of cross-platform or platform independent languages are Python, JavaScript, Java etc.

Java Libraries, Middle-ware, and Database options

- java.lang
- java.util
- java.sql
- java.io
- java.nio
- java.awt
- javax.swing

CODERINDEED

CSE310: Programming in Java

Fundamentals of Programming in Java

Naming Conventions

All the letters of an identifier in Java should be in lower case
Except:

Class Names: First letter of each word in the identifier should be capitalized.

Method Names: First letter of each word in the identifier should be capitalized except the first word.

Identifiers

A name in a program is called an identifier.

Identifiers can be used to denote classes, methods, variables, and labels.

An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters.

Example: number, Number, sum_$, bingo, $$_100

Note: Identifiers must not begin with a number.

Keywords

Keywords are reserved identifiers that are predefined in the language.

Cannot be used as names for a variable, class, or method.

All the keywords are in lowercase.

There are 50 keywords currently defined in the Java language.

The keywords **const** and **goto** are reserved but not used.

true, false, and **null** are also reserved.

Java Keywords

CODERINDEED

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

Writing Your First Java Program

```
class MyJavaProgram
    {
        public static void main(String args[])
            {
                System.out.println("Have fun in Java…");
            }
    }
```

Compiling and Executing Java Program

Step-1: Save your file with .java extension.

- Example: Program1.java

NOTE: If the class is public then the file name MUST BE same as the name of the class.

Step-2: Compile your .Java file using javac compiler from the location where the file is saved.

```
javac Program1.java
```

Compiling and Executing Java Program

Step-3: Execute your java class which contains the following method: public static void main(String args[]) {}

java MyJavaProgram

Access Specifiers

Specifier	Sub class (Same Package)	Non-Sub class (Same Package)	Sub class (Different Package)	Non-Sub class (Different Package)
public	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	No
default	Yes	Yes	No	No
private	No	No	No	No

Important Points

- A user defined outer class can be either public or default. We can not define a private or protected class.

- One file can have multiple classes/interfaces but out of them only one can be public.

CODERINDEED

CSE310: Programming in Java

Topic: Variables & their Scope

Outlines

[Expected Time: 1 Hours]

- Variables and their Scope
  - Local Variable
  - Instance Variable
  - Class Variable

Variables

- Variable are the data members or the fields defined inside a class.

- There are three types of variables:
  - Local Variable
  - Instance Variable
  - Class Variable

Local Variable

- Local variables are those data members which are declared in methods, constructors, or blocks.
- They are created only when the method, constructor or block is executed and destroyed as soon as the execution of that block is completed.
- So the visibility of local variables is inside the block only, they can’t be accessed outside the scope of that block.

Important

- Access modifiers are NOT ALLOWED with local variables.
- Local variables are created in STACK.
- Default value is NOT ASSIGNED to the local variables in Java.

Instance Variables

- Variables which are declared in a class, but outside a method, constructor or any block are known as instance variables.
- They are created inside the object in heap.
- Each object has its own set of instance variables and they can be accessed/modified separately using object reference.

- Instance variables are created when an object is created with the use of the 'new' keyword and destroyed when the object is destroyed.

- Access modifiers can be used with instance variables.

- Instance variables have default values.

- For numbers the default value is 0,
- for Booleans it is false
- and for object references it is null.

Class Variable

- Class variables are also known as static variables.
- Variable which are declared in a class, but outside a method, constructor or a block and qualified with ‘static’ keyword are known as class variables.

- Only one copy of each class variable is created, regardless of how many objects are created from it.

- Static variables can be accessed by calling with the class name.

ClassName.VariableName

- Static variables are created with the start of execution of a program and destroyed when the program terminates.
- Default values are same as instance variables.
- A public static final variable behaves as a CONSTANT in Java.
- Static variables can be initialized using static block also.

Variable Initialization

Local variables must be initialized explicitly by the programmer as the default values are not assigned to them where as the instance variables and static variables are assigned default values if they are not assigned values at the time of declaration.

Brainstorming 1

What will be the output of the following Program?

```
class VariableDemo
{
   public static void main(String [] rk)
    {
          public int x = 10;
         System.out.print(x);
    }
}
```

Brainstorming 2

What will be the output of the following Program?

```
class VariableDemo
{
   static int x;
   public static void main(String [] rk)
    {
          int x;
         System.out.print(x);
    }
}
```

Brainstorming 3

What will be the output of the following Program?

```
class VariableDemo
{
   static int x;
   public static void main(String [] rk)
    {
          int x;
         System.out.print(VariableDemo.x);
    }
}
```

CODERINDEED

CSE310: Programming in Java

Topic: Operators in Java

Outlines

[Expected Time: 1 Hours]

- Introduction
- Assignment Operator
- Arithmetic Operator
- Relational Operator
- Bitwise Operator
- Conditional Operator
- Type Comparison Operator
- Unary Operator

Introduction

- Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

Assignment Operator

- One of the most common operators is the simple assignment operator "=".
- This operator assigns the value on its right to the operand on its left.

Example:

```
int salary = 25000;        double speed = 20.5;
```

- This operator can also be used on objects to assign object references.

```
Student s = new Student();
Student s2 = s;
```

Arithmetic Operators

- Java provides operators that perform addition, subtraction, multiplication, and division.

Operator	Description
+	**Additive operator (also used for String concatenation)**
-	**Subtraction operator**
*	**Multiplication operator**
/	**Division operator**
%	**Remainder operator**

Compound Assignments

- Arithmetic operators are combined with the simple assignment operator to create compound assignments.
- Compound assignment operators are +=, -=, *=, /+, %=
- For example, x+=1; and x=x+1; both increment the value of x by 1.

Relational Operators

- Relational operators determine if one operand is greater than, less than, equal to, or not equal to another operand.

- It always returns boolean value i.e true or false.

Relational Operators

Operator	Description
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Type Comparison Operators

- The instanceof operator is used to compare an object to a specified type i.e. class or interface.

- It can be used to test if an object is an instance of a class or subclass, or an instance of a class that implements a particular interface.

Example

```
class Parent{}
class Child extends Parent{}
class TestInstanceOf  {
   public static void main(String [] rk){
          Parent p = new Parent();
          Child c = new Child();
          System.out.println(p instanceof Child);
          System.out.println(p instanceof Parent);
          System.out.println(c instanceof Parent);   }
    }
```

Unary Operators

- The unary operators require only one operand.

Operator	Description
+	Unary plus operator; indicates positive value
-	Unary minus operator; negates an expression
++	Increment operator; increments a value by 1
--	Decrement operator; decrements a value by 1
!	Logical complement operator; inverts the value of a boolean

CODERINDEED

Boolean Logical Operators

- The Boolean logical operators shown here operate only on boolean operands.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

- The following table shows the effect of each logical operation:

A	B	A | B	A & B	A ^ B	~ A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

CODERINDEED

Short-Circuit Logical Operators

- These are secondary versions of the Boolean AND and OR operators, and are known as short-circuit logical operators.

- OR operator results in true when A is true , no matter what B is. Similarly, AND operator results in false when A is false, no matter what B is.

- If we use the || and && forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.

- This is very useful when the right-hand operand depends on the value of the left one in order to function properly.

- For example

```
if (denom != 0 && num / denom > 10)
```

- The following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it.

The ? Operator

- Java includes a special ternary (three-way)operator, ?, that can replace certain types of if-then-else statements.
- The ? has this general form:

expression1 ? expression2 : expression3

- Here,expression1 can be any expression that evaluates to a boolean value.

- If expression1 is true , then expression2 is evaluated; otherwise, expression3 is evaluated.

- Both expression2 and expression3 are required to return the same type, which can’t be void.

int ratio = denom == 0 ? 0 : num / denom **;**

- When Java evaluates this assignment expression, it first looks at the expression to the left of the question mark.
- If denom equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? expression.
- If denom does not equal zero, then the expression after the colon is evaluated and used for the value of the entire ? expression.
- The result produced by the? operator is then assigned to ratio.

Bitwise Operators

These operators act upon the individual bits of their operands.

Can be applied to the integer types, long, int, short, char, and byte.

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

Bitwise Logical Operators

- The bitwise logical operators are
  - ~ (NOT)
  - & (AND)
  - | (OR)
  - ^ (XOR)

A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

CODERINDEED

The Left Shift Operator

- The left shift operator,<<, shifts all of the bits in a value to the left a specified number of times.

value << num

- Example:

01000001	65	<< 2
00000100	4	

The Right Shift Operator

- The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times.

value >> num

- It is also known as signed right shift.

- Example:

00100011	35 >> 2
00001000	8

- When we are shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit.

- This is called *sign extension and serves to preserve* the sign of negative numbers when you shift them right.

CODERINDEED

The Unsigned Right Shift

- In these cases, to shift a zero into the high-order bit no matter what its initial value was. This is known as an unsigned shift.

- To accomplish this, we will use Java's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit.

- Example:
  - 11111111 11111111 11111111 11111111 –1 in binary as an int
  - >>>24
  - 00000000 00000000 00000000 11111111 255 in binary as an int

CODERINDEED

Operator Precedence

Highest			
( )	**[]**	**.**	
++	--	**%**	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	**Op=**		
Lowest			

Brainstorming 1

What will be the output of the following code snippets?

- byte b = 30; System.out.println(~b);
- byte b = -53; System.out.println(~b);
- System.out.println(34>>3);
- System.out.println(-34>>3);
- System.out.println(-34>>>3);

Brainstorming 2

```
int x = 10, y = 5;
while(x- -  > 7 || + + y < 8 );
System.out.print(x);
System.out.print(y);
```

A. 95
B. 67
C. 78
D. 48
E. 77
F. N.O.T

Brainstorming 3

```
System.out.print(2>1||4>3?false:true);
class X{}
class Y extends X{}
class Z extends Y{}

X x1 = new Y();
Y y1 = new Z();
Y y2 = new Y();
System.out.println( x1 instanceof  X);
System.out.println( x1 instanceof  Z);
System.out.println( y1 instanceof  Z);
System.out.println( y2 instanceof  X);
```

Brainstorming 4

```
System.out.print(2>1||4>3?false:true);
```

CODERINDEED

Programming in Java

Topic: Control Flow Statements

SELECTION STATEMENTS

- Java supports two selection statements: **if** and **switch.**

if statement

```
if (condition) statement1;
else statement2;
```

- Each statement may be a single statement or a compound statement enclosed in curly braces (block).
- The condition is any expression that returns a **boolean value.**
- **The else** clause is optional.
- If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.
- In no case will both statements be executed.

Nested ifs

- A nested if is an if statement that is the target of another if or else.
- In nested ifs an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

```
if (i == 10) {
                if (j < 20) a = b;
                if (k > 100) c = d;       // this if is
                else a = c;               // associated with this else
                }
else a = d;                               // this else refers to if(i == 10)
```

CODERINDEED

The if-else-if Ladder

- A sequence of nested ifs is the if-else-if ladder.

```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
...
else
  statement;
```

- The if statements are executed from the top to down.

switch

- The switch statement is Java's multi-way branch statement.
- provides an easy way to dispatch execution to different parts of your code based on the value of an expression.
- provides a better alternative than a large series of **if-else-if statements.**

```
switch (expression) {
        case value1:
                // statement sequence
                break;
        case value2:
                // statement sequence
                break;
        ...
        case valueN:
                // statement sequence
                break;
        default:
                // default statement sequence
                }
```

- The expression must be of type byte, short, int, char, String or enum.
- Each of the values specified in the case statements must be of a type compatible with the expression.
- Each case value must be a unique literal (i.e. constant not variable).
- Duplicate case values are not allowed.
- The value of the expression is compared with each of the literal values in the case statements.
- If a match is found, the code sequence following that case statement is executed.
- If none of the constants matches the value of the expression, then the default statement is executed.
- The default statement is optional.

- If no case matches and no default is present, then no further action is taken.
- The break statement is used inside the switch to terminate a statement sequence.
- When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement.

```
class SampleSwitch {
    public static void main(String args[]) {
        for(int i=0; i<6; i++)
            switch(i) {
                case 0:
                    System.out.println("i is zero.");
                    break;
                case 1:
                    System.out.println("i is one.");
                    break;
                case 2:
                    System.out.println("i is two.");
                    break;
                default:
                    System.out.println("i is greater than 2.");
            }
        }
    }
```

Nested switch Statements

- When a switch is used as a part of the statement sequence of an outer switch. This is called a nested switch.

```
switch(count) {
        case 1:
                switch(target) { // nested switch
                        case 0:
                                System.out.println("target is zero");
                                break;
                        case 1: // no conflicts with outer switch
                                System.out.println("target is one");
                                break;
                              }
                        break;
        case 2: // ...
```

Difference between ifs and switch

- switch can only test for equality, whereas if can evaluate any type of Boolean expression. That is, the switch looks only for a match between the value of the expression and one of its case constants.

- A switch statement is usually more efficient than a set of nested ifs.

- Note: No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.

Let's do Something…

Write a menu driven program which prompts the user to enter

[1. Saving Account

2. Current Account]

Ask the user to enter the age. If the age is less than 18 then print "current account can not be opened" and if the age is less than 21 but user is eligible for current account then print "saving account can not be opened but you can open current account". Otherwise print "You are not eligible for Saving Account".

If the user is eligible for given type of account then print "You are eligible to open the CURRENT/SAVING account"

ITERATION STATEMENTS (LOOPS)

CODERINDEED

Iteration Statements

- In Java, iteration statements (loops) are:
  - for
  - while, and
  - do-while
  - for each (Enhanced For Loop)

- A loop repeatedly executes the same set of instructions until a termination condition is met.

While Loop

- While loop repeats a statement or block while its controlling expression is true.
- The condition can be any Boolean expression.
- The body of the loop will be executed as long as the conditional expression is true.
- When condition becomes false, control passes to the next line of code immediately following the loop.

```
while(condition)
  {
    // body of loop
  }
```

```
class While
    {
        public static void main(String args[]) {
        int n = 10;
        char a = 'G';
        while(n > 0)
                    {
                      System.out.print(a);
                      n--;
                      a++;
                    }
           }
    }
```

- The body of the loop will not execute even once if the condition is false.

- The body of the while (or any other of Java's loops) can be empty. This is because a null statement (one that consists only of a semicolon) is syntactically valid in Java.

do-while

- The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

```
do {
        // body of loop
} while (condition);
```

- Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression.
- If this expression is true, the loop will repeat. Otherwise, the loop terminates.

for Loop

```
for (initialization; condition; iteration)
    {
     // body
    }
```

- Initialization portion sets the value of loop control variable.
- Initialization expression is only executed once.
- Condition must be a Boolean expression. It usually tests the loop control variable against a target value.
- Iteration is an expression that increments or decrements the loop control variable.

The for loop operates as follows.

- When the loop first starts, the initialization portion of the loop is executed.

- Next, condition is evaluated. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.

- Next, the iteration portion of the loop is executed.

```
class ForTable
        {
                public static void main(String args[])
                 {
                        int n;
                        int x=5;
                        for(n=1; n<=10; n++)
                          {
                           int p = x*n;
                           System.out.println(x+"*"+n +"="+ p);
                          }
                }
        }
```

What will be the output?

```
class Loop
    {
        public static void main(String args[])
            {
                for(int i=0; i<5; i++);
                        System.out.println (i++);
             }
     }
```

Declaring loop control variable inside loop

- We can declare the variable inside the initialization portion of the for.

```
for ( int i=0; i<10; i++)
        {
           System.out.println(i);
        }
```

- Note: The scope of this variable i is limited to the for loop and ends with the for statement.

CODERINDEED

Using multiple variables in a for loop

- More than one statement in the initialization and iteration portions of the for loop can be used.

Example 1:

```
class var2 {
        public static void main(String arr[]) {
                int a, b;
                b = 5;
                for(a=0; a<b; a++) {
                        System.out.println("a = " + a);
                        System.out.println("b = " + b);
                        b--;
                }
        }
}
```

- Comma (separator) is used while initializing multiple loop control variables.

Example 2:

```
class var21
        {
                public static void main(String arr[]) {
                        int x, y;
                        for(x=0, y=5; x<=y; x++, y--) {
                                System.out.println("x=  " +   x);
                                System.out.println("y =  " +  y);
                        }
                }
        }
```

- Initialization and iteration can be moved out from for loop.

Example 3:

```
class Loopchk
{
public static void main(String arr[])
  {
   for(int i=1, j=5; i>0 && j>2; i++, j--)
            System.out.println("i is: "+ i + "and j is: "+j);

   }
 }
```

CODERINDEED

For-Each Version of the for Loop

- Beginning with JDK 5, a second form of for was defined that implements a “for-each” style loop.
- For-each is also referred to as the **enhanced for loop.**
- Designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to end.

for (type itr-var : collection) statement-block

- *type* specifies the type.
- *itr-var* specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end.
- The collection being cycled through is specified by collection.
- With each iteration of the loop, the next element in the collection is retrieved and stored in itr-var.
- The loop repeats until all elements in the collection have been obtained.

```
class ForEach
    {
     public static void main(String arr[])
            {
                int num[] = { 1, 2, 3, 4, 5 };
                int sum = 0;
                for(int i : num)
                    {
                        System.out.println("Value is: " + i);
                        sum += i;
                    }
                System.out.println("Sum is:  " + sum);
            }
    }
```

CODERINDEED

Iterating Over Multidimensional Arrays

```
class ForEachMArray {
        public static void main(String args[]) {
                int sum = 0;
                int num[][] = new int[3][5];
        // give num some values
                for(int i = 0; i < 3; i++)
                        for(int j=0; j < 5; j++)
                                num[i][j] = (i+1)*(j+1);
        // use for-each for to display and sum the values
                for(int x[] : num) {
                        for(int y : x) {
                                System.out.println("Value is: " + y);
                                sum += y;
                                    }
                                }
                System.out.println("Summation: " + sum);
                        }
                }
```

coderindeed.in

Nested Loops

CODERINDEED

```
class NestedLoop
    {
        public static void main(String arr[])
                {
                    int i, j;
                    for(i=0; i<10; i++)
                            {
                                for(j=i; j<10; j++)
                                    System.out.print("* ");
                                System.out.println( );
                                }
                }
    }
```

CODERINDEED

CSE310: Programming in Java

Topic: Branching Statements

Outlines

[Expected Time: 1 Hours]

- break Statement
- continue Statement
- return Statement

break Statement

- break statement has three uses:
  - terminates a statement sequence in a switch statement
  - used to exit a loop
  - used as a “civilized” form of goto.

- The break statement has two forms:
  - labeled
  - unlabeled.

Unlabeled break

- An unlabeled break is used to terminate a for, while, or do-while loop and switch statement.

Example 1:

```
public void breakTest()
    {
        for(int i=0; i<100; i++)
          {
              if(i == 10) break;
              System.out.println("i: " + i);
          }
        System.out.println("Loop completed");
    }
```

Example

```
public void switchTest() {
    for(int i=0; i<5; i++)
        switch(i) {
            case 0:
                System.out.println("i is zero.");        break;
            case 1:
                System.out.println("i is one.");         break;
            case 2:
                System.out.println("i is two.");         break;
            default:
                System.out.println("i is greater than 2.");
            }
        }
```

Labeled break Statement

- Java defines an expanded form of the break statement.

break *label;*

- By using this form of break, we can break out of one or more blocks of code.

- When this form of break executes, control is transferred out of the named block.

Example

CODERINDEED

```
class BreakDemo{
   public static void main(String [] rk)
     {
        outer:
            for(int i=0; i<3; i++){
                        System.out.println("Outer "+ i);
                        inner:
                                    for(int j=0; j<3; j++)
                                    {
                                                System.out.println("Inner "+j);
                                                if(i== j+1)
                                                     break outer;
                                                System.out.println("Bye");
                                    }
                        }
     }
}
```

NOTE

- The break statement terminates the labeled statement; it does not transfer the flow of control to the label.

- Control flow is transferred to the statement immediately following the labeled (terminated) statement.

continue Statement

- The continue statement skips the current iteration of a for, while , or do-while loop.

- The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop.

CODERINDEED

Example

```
class ContinueDemo {
    public static void main(String[] rk) {
        String str = “she saw a ship in the sea”;
        int size = str.length();
        int count = 0;
        for (int i = 0; i < size; i++)
            {
                if (str.charAt(i) != ‘s’)
                    continue;
                count++;
            }
        System.out.println(“Number of s in “+ str + “ = ”+ count); } }
```

Labeled continue Statement

- A labeled continue statement skips the current iteration of an outer loop marked with the given label.

CODERINDEED

Example

```
class ContinueLabel {
    public static void main(String [] rk) {
        outer: for (int i=0; i<3; i++) {
                    for(int j=0; j<3; j++) {
                        if(j > i) {
                            System.out.println(“Hi”);
                            continue outer;   }
                        System.out.print(" " + (i * j));
                     }
                }
        System.out.println(“Done”);   }     }
```

return Statement

- The return statement exits from the current method, and control flow returns to where the method was invoked.

- The return statement has two forms: one that returns a value, and one that doesn't.

- To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

```
return ++count;
```

return Statement

- The data type of the returned value must match the type of the method's declared return value.

- When a method is declared void, use the form of return that doesn't return a value.

```
return;
```

Brainstorming 1

CODERINDEED

```
public static void main(String [] rk){
    outer:
        for(int i=0; i<3; i++)
          {
        inner:
            for(int j=0; j<3; j++)
                    {
                      System.out.println(i + ", "+ j);
                      if(j==2)        break inner;
                      if(i==j)        continue outer;
                    }
           System.out.println("Bye");
           }
}
```

Brainstorming 2

```
class BreakTest{
   public static void main(String [] rk)
    {
      hello:
          for(int a=1; a<3; a++)
             System.out.print("Hello");
          int i = 1;
          if(i==1)
                    break hello;
          System.out.print("Not reachable");
    }
}
```

Brainstorming 3

CODERINDEED

```
public static void main(String [] rk)
    {  int n=5;
       outer:  for(int a=1; a<5; a++)
                    {        int i=0, j=0;     System.out.println();

                    space:  while(true)  {
                                    System.out.print(" ");          i++;
                                    if(i==n-a)       break space;   }

                    star:     while(true)  {
                                    System.out.print(" * ");        j++;
                                    if(j==a) continue outer;
                                    }
                    }
    }
```

CODERINDEED

CSE310: Programming in Java

Topic: Array and Enum

Outlines

[Expected Time: 2 Hours]

- Introduction
- Array Creation
- Array Initialization
- Array as an Argument
- Array as a return type
- Enumerations

Array

- Definition:

  An array is a finite collection of variables of the same type that are referred to by a common name.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index (subscript).
- Array elements are stored in contiguous memory locations.

- Examples:
  - **C**ollection of numbers
  - **C**ollection of names

CODERINDEED

Examples

Array of numbers:

10	23	863	8	229

Array of names:

Sam	Shanu	Riya

Array of suffixes:

ment	tion	ness	ves

CODERINDEED

One-Dimensional Arrays

- A one-dimensional array is a list of variables of same type.
- The general form of a one-dimensional array declaration is:

type [] var-name; array-var = new type[size];

OR

type [] var-name = new type[size];

Example:

```
int [] num =  new int [10];
```

Syntax

Declaration of array variable:

data-type variable-name[];

eg. *int marks[];*

This will declare an array named ‘marks’ of type ‘int’. But no memory is allocated to the array.

Allocation of memory:

variable-name = new data-type[size];

eg. *marks = new int[5];*

This will allocate memory of 5 integers to the array ‘marks’ and it can store upto 5 integers in it. ‘new’ is a special operator that allocates memory.

Accessing elements in the array:

• Specific element in the array is accessed by specifying name of the array followed the index of the element.

• All array indexes in Java start at zero.

variable-name[index] = value;

Example:

marks[0] = 10;

This will assign the value 10 to the 1st element in the array.

marks[2] = 863;

This will assign the value 863 to the 3rd element in the array.

Example

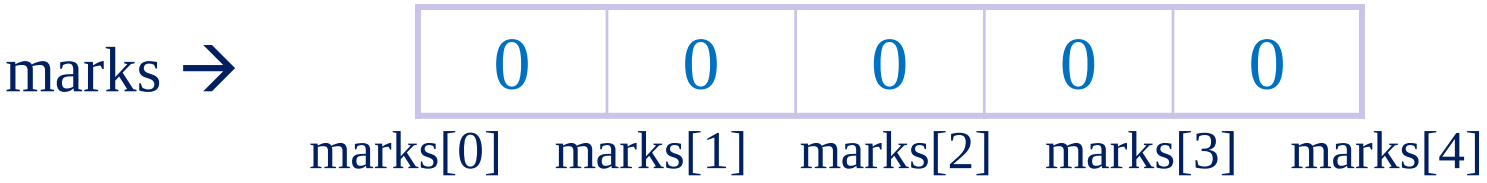
STEP 1 : (Declaration)

int marks[];

marks → null

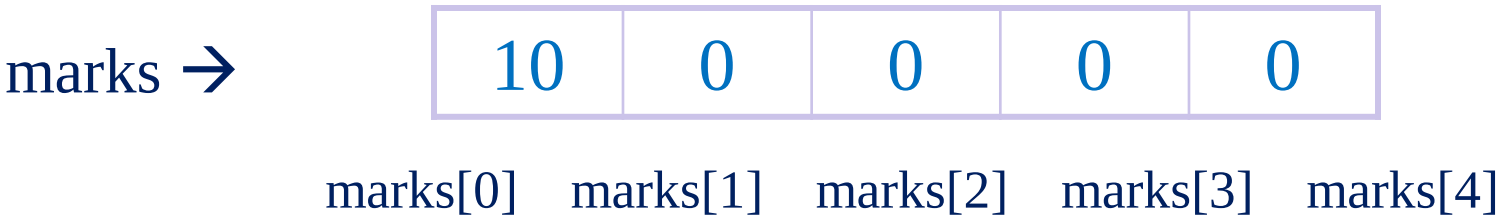
STEP 2: (Memory Allocation)

marks = new int[5];



STEP 3: (Accessing Elements)

marks[0] = 10;



- Size of an array can't be changed after the array is created.

- Default values:
  - zero (0) for numeric data types,
  - \u0000 for chars and
  - false for Boolean types
  - null for references

- Length of an array can be obtained as:

array_ref_var.length

CODERINDEED

Example

```
class Demo_Array
    {
        public static void main(String args[])
                {
                        int marks[];
                        marks = new int[3];

                        marks[0] = 10;
                        marks[1] = 35;
                        marks[2] = 84;

    System.out.println("Marks of 2nd student=" + marks[1]);

                }
    }
```

Note

- Arrays can store elements of the ***same data type***. Hence an ***int*** array CAN NOT store an element which is not an int.

- Though an element of a compatible type can be converted to int and stored into the int array.

eg. marks[2] = (int) 22.5;

This will convert '22.5' into the int part '22' and store it into the 3rd place in the int array 'marks'.

- Array indexes start from zero. Hence 'marks[index]' refers to the (index+1)th element in the array and 'marks[size-1]' refers to last element in the array.

Array Initialization

CODERINDEED

1. data Type [] array_ref_var = {value0, value1, …, value n};

2. data Type [] array_ref_var;
   array_ref_var = {value0, value1, …,value n};

3. data Type [] array_ref_var = new data Type [n+1];

   array_ref_var [0] = value 0;

   array_ref_var [1] = value 1;

   …

   array_ref_var [n] = value n;

Exercise

Write a program which prompts the user to enter the number of subjects.

Now read the marks of all the subjects from the user using Scanner class.

Write a method which calculates the percentage of the user.

Multi-Dimensional Array

- Multidimensional arrays are arrays of arrays.

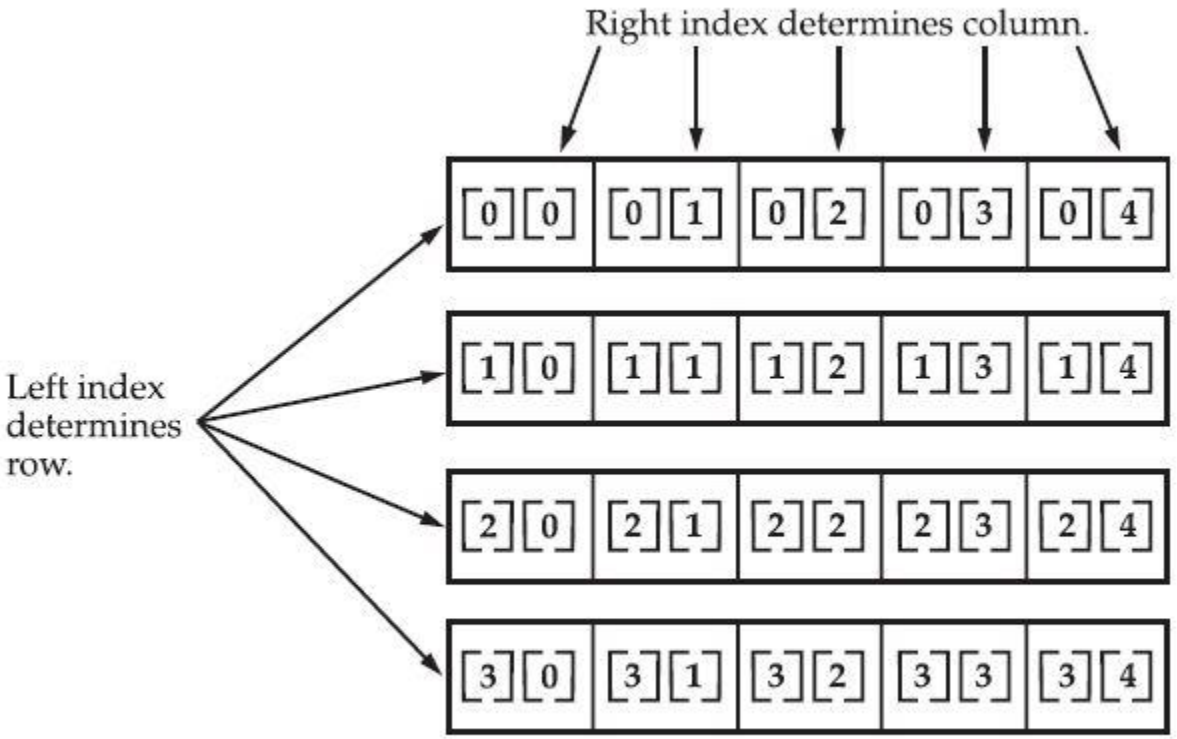
- Two-Dimensional arrays are used to represent a table or a matrix.

- Creating Two-Dimensional Array:

```
int twoD[][] = new int[4][5];
```

CODERINDEED

Conceptual View of 2-Dimensional Array



Given: int twoD [ ] [ ] = new int [4] [5] ;

```
class TwoDimArr
    {
      public static void main(String args[])
            {
              int twoD[][]= new int[4][5];
              int i, j, k = 0;
              for(i=0; i<4; i++)
                    for(j=0; j<5; j++)
                        {
                          twoD[i][j] = k;
                          k++;
                        }
                    for(i=0; i<4; i++)
                        {
                    for(j=0; j<5; j++)
                          System.out.print(twoD[i][j] + " ");
                          System.out.println();
                        }
             }
    }
```

- When we allocate memory for a multidimensional array, we need to only specify the memory for the first (leftmost) dimension.

```
int twoD[][] = new int[4][];
```

- The other dimensions can be assigned manually.

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
    public static void main(String args[]) {
        int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];
            int i, j, k = 0;
        for(i=0; i<4; i++)
            for(j=0; j<i+1; j++)
                {
                  twoD[i][j] = k;
                  k++;
                }
            for(i=0; i<4; i++) {
                for(j=0; j<i+1; j++)
                        System.out.print(twoD[i][j] + " ");
                        System.out.println();
                    }
            }
        }
```

CODERINDEED

Initializing Multi-Dimensional Array

```
class Matrix {
    public static void main(String args[]) {
        double m[][] = {
                                { 0*0, 1*0, 2*0, 3*0 },
                                { 0*1, 1*1, 2*1, 3*1 },
                                { 0*2, 1*2, 2*2, 3*2 },
                                { 0*3, 1*3, 2*3, 3*3 }
                        };
        int i, j;
        for(i=0; i<4; i++) {
                for(j=0; j<4; j++)
                        System.out.print(m[i][j] + " ");
                System.out.println();
                        }
                }
        }
```

CODERINDEED

Alternative Array Declaration

type[ ] var-name;

- Example:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

- This alternative declaration form offers convenience when declaring several arrays at the same time.

- Example: int[] nums, nums2, nums3;

Array Cloning

- To actually create another array with its own values, Java provides the **clone()** method.

- arr2 = arr1; (assignment)

  is not equivalent to

  arr2 = arr1.clone(); (cloning)

  In first case, Only one array is created and two references arr1 and arr2 are pointing to the same array. While in second case two different arrays are created.

Array as Argument

- Arrays can be passed as an argument to any method.

Example:

```
void show ( int []x) {}

public static void main(String arr[]) {}

void compareArray(int [] a, int [] b ){}
```

Array as a return Type

- Arrays can also be returned as the return type of any method

Example:

```
public int [] Result (int roll_No, int marks )
```

Assignment for Practice

- WAP in which use a method to convert all the double values of argumented array into int.

  public int[] double_To_Int(double [] ar)

- WAP to create a jagged Array in which rows and columns should be taken as input from user.

Assignment for Practice

- WAP in which prompt the user to enter the number of subjects and number of CA in each subject. Read the marks of each CA and store in a two dimensional array.

  Implement a method

  public char[] findGrades(double [][] marks)

Variable Length Argument List

- Used to pass variable number of arguments of the same type to a method.

Syntax: int … arr

Example: Consider the following method

public void add(int [] arr)

In this method user has to pass an array of int type. But if we use

public void add(int ... arr)

Then One can invoke the same method in different ways:

a. add(3, 4, 5);

b. int [] x = {3, 4, 5, 6, 7}; add(x);

JAVA ENUM

Introduction

- Enum in java is a data type that contains fixed set of constants.
- It can be thought of as a class having fixed set of constants.
- The java enum constants are static and final implicitly. It is available from JDK 1.5.
- It can be used to declare days of the week, Months in a Year etc.

Advantages of Enum

- enum improves type safety
- enum can be easily used in switch
- enum can be traversed
- enum can have fields, constructors and methods
- enum may implement many interfaces but cannot extend any class because it internally extends Enum class

values() method

- The java compiler internally adds the values() method when it creates an enum.
- The values() method returns an array containing all the values of the enum.

Important

- Enum can not be instantiated using new keyword because it contains private constructors only.
- We can have abstract methods and can provide the implementation of these methods in an Enum.
- The enum can be defined within or outside the class because it is similar to a class.

Example

```
class EnumDemo
   {
    public enum Season { SUMMER,  WINTER, SPRING}

    public static void main(String[] rk)
           {
                  for (Season s : Season.values())
                         System.out.println(s);
           }
   }
```

Example 2

```
public enum Season { SUMMER,  WINTER, SPRING}

class EnumDemo2
   {
    public static void main(String[] rk)
           {
                  for (Season s : Season.values())
                         System.out.println(s);
           }
   }
```

CODERINDEED

Enum with Constructor

```
public enum MonthWithDays {
        January(31),  February(28), March(31);
        int days;
        MonthWithDays(int a) { days = a; }
}

class EnumDemo3
  {
   public static void main(String[] rk)
        {
              for (MonthWithDays s : MonthWithDays.values())
                    System.out.println(s);
        }
  }
```

Brainstorming 1

- Can You declare an enum private?
- Can you define an enum inside any block in a class?
- Can you define multiple enums inside same class?

Let’s Do Something

Write a program which accepts an array of marks (N subjects) containing double type values and return the array of marks after rounding up the marks of all the subjects.

Let's Do Something

Write a program which prompts the user to enter the number of subjects and their names and reads the number of CAs of all the subjects. Read the marks of all the CAs for each subject.

Now implement a method :

public double [] percentage (double [] [] marks)

which returns the percentage of all the subjects. Display the percentage of each subject with name of subject.

CODERINDEED

Programming in Java

Topic: Wrapper Classes

Wrapper Class

- Wrapper classes are classes that allow primitive types to be accessed as objects.
- Wrapper class in java provides the mechanism to convert *primitive into object* and *object into primitive*.
- Wrapper class is wrapper around a primitive data type because they "wrap" the primitive data type into an object of that class.

Wrapper Classes

- Each of Java's eight primitive data types has a class dedicated to it.
- They are one per primitive type: Boolean, Byte, Character, Double, Float, Integer, Long and Short.
- Wrapper classes make the primitive type data to act as objects.

Primitive Data Types and Wrapper Classes

Data Type	Wrapper Class
byte	Byte
short	Short
int	***Integer***
long	Long
char	***Character***
float	Float
double	Double
boolean	Boolean

Why Wrapper Class?

- Most of the objects collection store objects and not primitive types.
- Primitive types can be used as object when required.
- As they are objects, they can be stored in any of the collection and pass this collection as parameters to the methods.

What is the need of Wrapper Classes?

- Wrapper classes are used to be able to use the primitive data-types as objects.

- Many utility methods are provided by wrapper classes.

  To get these advantages we need to use wrapper classes.

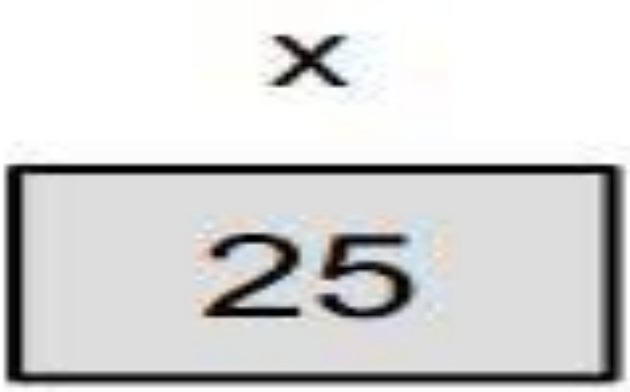
Difference b/w Primitive Data Type and Object of a Wrapper Class

- The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

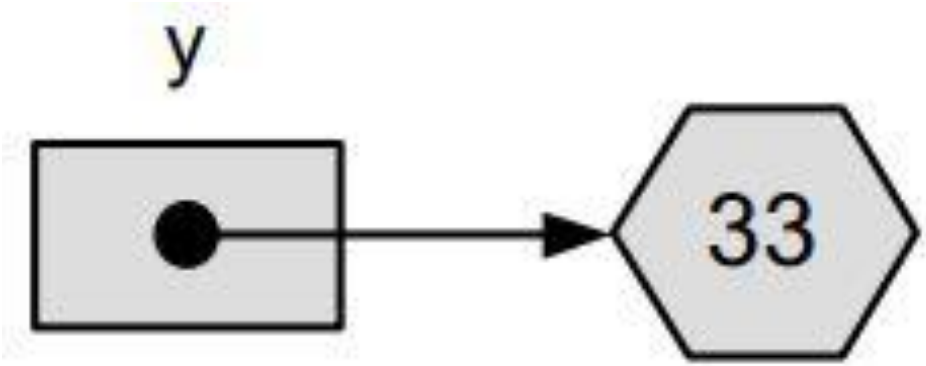
```
int x = 25;

Integer y = new Integer(33);
```

- The first statement declares an int variable named x and initializes it with the value 25.



- The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.



- Clearly x and y differ by more than their values:

  x is a variable that holds a value;

  y is an object variable that holds a reference to an object.

Boxing and Unboxing

- The wrapping is done by the compiler.
- if we use a primitive where an object is expected, the compiler boxes the primitive in its wrapper class.
- Similarly, if we use a number object when a primitive is expected, the compiler un-boxes the object.

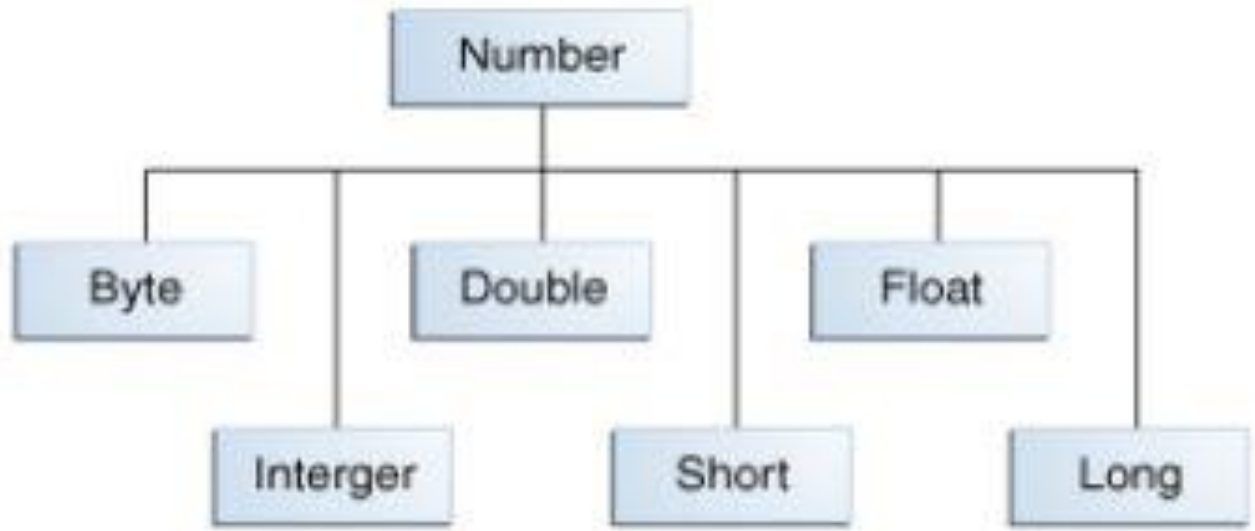
Example of boxing and unboxing:

- Integer x, y; x = 12; y = 15; System.out.println(x+y);
- When x and y are assigned integer values, the compiler boxes the integers because x and y are integer objects.
- In the println() statement, x and y are unboxed so that they can be added as integers.

Numeric Wrapper Classes

- All of the numeric wrapper classes are subclasses of the abstract class Number .

- All of them implements Comparable interface.



Features of Numeric Wrapper Classes

- All the numeric wrapper classes provide a method to convert a numeric *string into a primitive value*.

public static type parseType (String Number)

- parseInt()
- parseFloat()
- parseDouble()
- parseLong()

…

Features of Numeric Wrapper Classes

- All the wrapper classes provide a static method toString to provide the *string representation of the primitive values*.

public static String toString (type value)

Example:

public static String toString (int a)

Features of Numeric Wrapper Classes

- All numeric wrapper classes have a static method valueOf, which is used to *create a new object initialized to the value* represented by the specified string.

public static DataType valueOf (String s)

Example:

```
Integer i = Integer.valueOf (“135”);
Double d = Double.valueOf (“13.5”);
```

Methods implemented by subclasses of Number

- Compares this Number object to the argument.

  int compareTo(Byte anotherByte)
  int compareTo(Double anotherDouble)
  int compareTo(Float anotherFloat)
  int compareTo(Integer anotherInteger)
  int compareTo(Long anotherLong)
  int compareTo(Short anotherShort)

- returns int after comparison (-1, 0, 1).

Methods implemented by subclasses of Number

boolean equals(Object obj)

- Determines whether this number object is equal to the argument.
- The methods return true if the argument is not null and is an object of the same type and with the same numeric value.

Character Class

- Character is a wrapper around a char.

- The constructor for Character is :

  Character(char ch)

  Here, ch specifies the character that will be wrapped by the Character object being created.

- To obtain the char value contained in a Character object, call charValue( ), shown here:

  char charValue( );

- It returns the encapsulated character.

Boolean Class

- Boolean is a wrapper around boolean values.

- It defines these constructors:

  Boolean(boolean boolValue)

  Boolean(String boolString)

- In the first version, boolValue must be either true or false.

- In the second version, if boolString contains the string “true” (in uppercase or lowercase i.e TrUE, trUE), then the new Boolean object will be true. Otherwise, it will be false.

- To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:

  boolean booleanValue( )

- It returns the boolean equivalent of the invoking object.

CODERINDEED

Programming in Java

Topic: ArrayList

By

coderindeed.in

Introduction

- Standard Java arrays are of a fixed length.
- After arrays are created, they cannot grow or shrink, which means we can not resize Arrays.
- Arrays are useful when we know in advance how many elements the array is to hold.
- ArrayList is a collection which provides the implementation of resizable array.

ArrayList

- The ArrayList class extends *AbstractList* and implements the *List* interface.
- Defined in *java.util* package.
- ArrayList supports dynamic arrays that can grow as needed.
- Array lists are created with an initial size.
- When this size is exceeded, the collection is automatically enlarged.
- When objects are removed, the array may be shrunk.

ArrayList Constructors

- T he ArrayList class supports three constructors.

➓ ArrayList() : creates an empty array list with an initial capacity sufficient to hold 10 elements.

➓ ArrayList(int capacity) : creates an array list that has the specified initial capacity.

➓ ArrayList(Collection c) : creates an array list that is initialized with the elements of the collection c.

Methods of ArrayList

boolean add(Object o)

- Appends the specified element to the end of this list.

void add(int index, Object element)

- Inserts the specified element at the specified position index in this list.
- Throws IndexOutOfBoundsException if the specified index is out of range.

boolean addAll(Collection c)

- Appends all of the elements in the specified collection to the end of this list.
- Throws NullPointerException if the specified collection is null.

boolean addAll(int index, Collection c)

- Inserts all of the elements of the specified collection into this list, starting at the specified position.
- Throws NullPointerException if the specified collection is null.

void clear()

- Removes all of the elements from this list.

Object remove(int index)

- Removes the element at the specified position in this list.

boolean remove(Object o)

- Removes the first specified object present in the arraylist.

int size()

- Returns the number of elements in the list.

boolean contains(Object o)

- Returns true if this list contains the specified element.

Object get(int index)

- Returns the element at the specified position in this list.

int indexOf(Object o)

- Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain the element.

int lastIndexOf(Object o)

- Returns the index in this list of the last occurrence of the specified element, or -1.

Let's Do Something…

Write a program to store the Employee objects in an arraylist such that each employee is having name, emp_id and salary attributes.

Implement a method getEmployee(String name) and display the emp_id and salary of that Employee.

CODERINDEED

Programming in Java

Inheritance (Method Overloading and Overriding)

By

coderindeed.in

Contents

- Abstract Class
- Inheritance
- Method Overloading
- Method Overriding
- Static and Dynamic Binding

Introduction

- Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes.
- In the Java programming language, each class is allowed to have one direct super-class, and each super-class has the potential for an unlimited number of subclasses.
- Inheritance is a fundamental object-oriented design technique used to create and organize reusable classes.

Abstract Class

- An *abstract class* is a class that is declared abstract.
- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

- Abstract class may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be sub-classed.

- If a class includes abstract methods, the class itself *must* be declared abstract, as in:

```
public abstract class GraphicObject
        {       // declare fields
                // declare non-abstract methods
                abstract void draw();
        }
```

NOTE:

- When an abstract class is sub-classed, the subclass usually provides implementations for all of the abstract methods in its parent class.
- However, if it does not, the subclass must also be declared abstract.

- It can be considered as a blank form consisting of several sections, some of them may be optional (non-abstract method) and some may be mandatory (abstract)…

- Can we Override a constructor?

  If yes, How? if No, Why?

Inheritance

- Inheritance defines *an is-a relationship* between a super-class and its subclasses. It means that the subclass (child) is a more specific version of the super-class (parent).

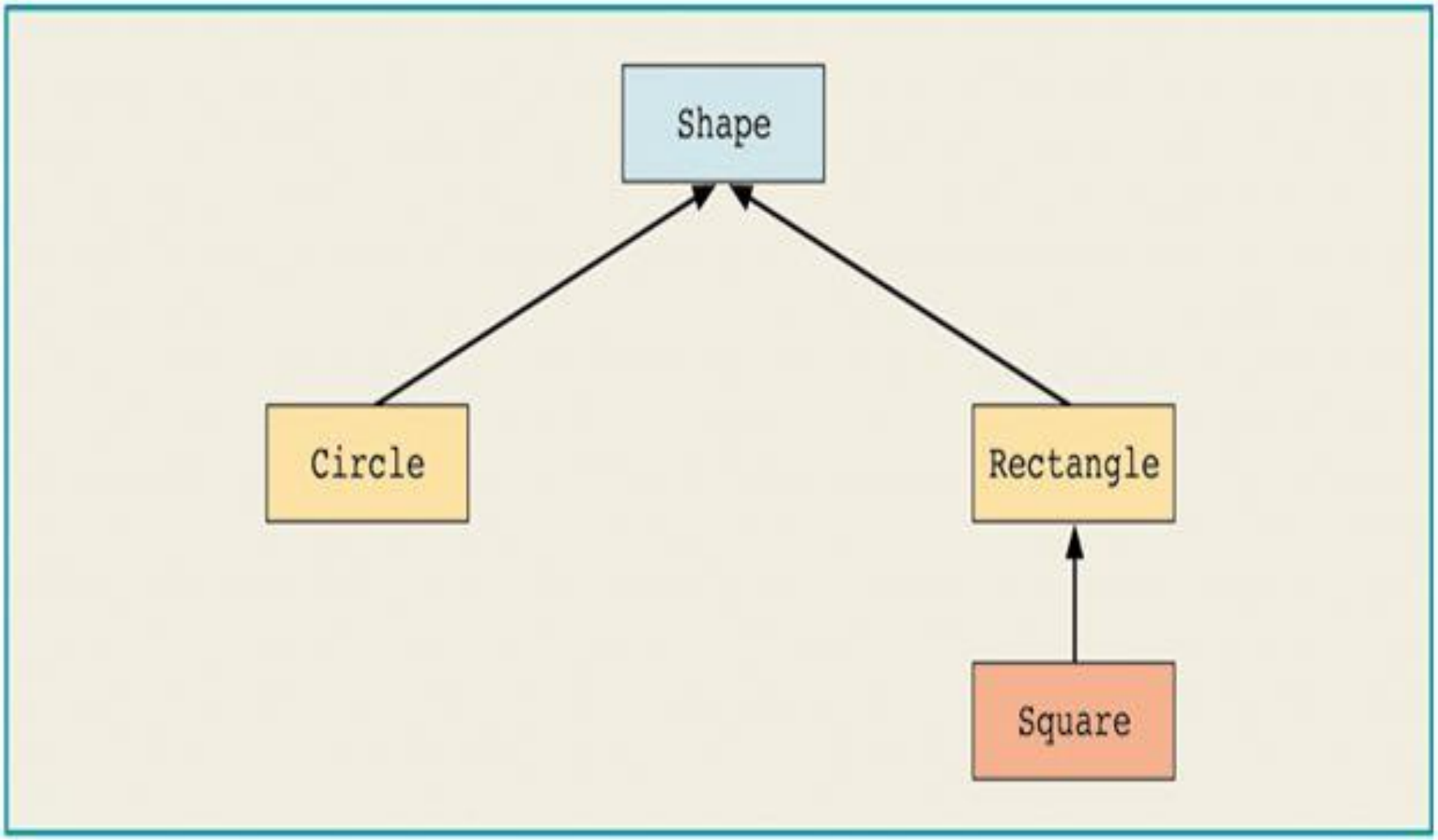
- An object of a subclass can be used wherever an object of the super-class can be used.

- Inheritance is used to build new classes from existing classes.

- The inheritance relationship is transitive: if class y extends class x, and a class z extends class y, then z will also inherit from class x.

Inheritance

- In Java, a class that is inherited is called a super-class.
- The class that does the inheriting is called a subclass.
- A Subclass inherits all of *the instance variables and methods* defined by the super-class and adds its own, unique elements.


Shape
Circle
Rectangle
Square

CODERINDEED

Inheritance Example

```
class Shape {
        int area(){…}
    }

class Rectangle extends Shape{
        int area() { area = length * width;}
        int length;          int width;
    }

class Square extends Rectangle  {
        int area() {…}
        int length;          int width = length;
    }
```

Key Points

- Private members of the super-class are inherited but they can not be accessed by the subclass directly.
- Members that have default accessibility in the super-class are also not inherited by subclasses in other packages.
- Constructors and initializer blocks are not inherited by a subclass.
- A subclass can extend only one super-class.

Types of Inheritance

The following kinds of inheritance are there in java.

- Simple Inheritance
- Multilevel Inheritance

- Simple Inheritance: A subclass is derived simply from it's parent class.

- There is only a sub class and it's parent class. It is also called single inheritance or one level inheritance.

- Multi-level Inheritance: A subclass is derived from a derived class.

Multiple Inheritance

- The mechanism of inheriting the features of more than one base class into a single class is known as multiple inheritance.
- Java does not support multiple inheritance till jdk 1.7 but the multiple inheritance can be achieved by using the interface.
- In Java Multiple Inheritance can be achieved through use of Interfaces by implementing more than one interfaces in a class.

Method Overloading

CODERINDEED

- Method overloading means having two or more methods with the same name but different signatures in the same scope.
- These two methods may exist in the same class or one in base class and another in derived class.
- It allows creating several methods with the same name which differ from each other in the type of the input and the output of the method.
- It is simply defined as the ability of one method to perform different tasks.

Example

```
class Area11
   {
      void area(int a)
         {
            int area = a*a;
            System.out.println("area of square is:" + area);
         }
      void area (int a, int b)
         {
            int area = a*b;
            System.out.println("area of rectangle is:" + area);
         }
   }
```

```
class OverloadDemo
   {
     public static void main (String arr[])
           {
            Area11 ar= new Area11();
            ar.area(10);
            ar.area(10,5);
           }
   }
```

Method Overriding

- Method overriding means having a **different implementation of the same method** in the **inherited class**.

- These two methods would have the **same signature, but different implementation**.

- One of these would exist in the **base class** and another in the **derived class**. These cannot exist in the same class.

- The version of a method that is executed will be determined by the object that is used to invoke it.

- If an object of a parent class is used to invoke the method, then the version in the parent class will be executed.

- If an object of the subclass is used to invoke the method, then the version in the child class will be executed.

```
class Override
    {
      public void display()
            {
              System.out.println("Hello...This is superclass display");
            }
    }

class Override1 extends Override
    {
      public void display()
            {
              System.out.println("Hi...This is overriden method in subclass");
            }
    }
```

```
class OverrideDemo
  {
    public static void main(String arr[])
        {
          Override o = new Override();
              o.display();
          Override1 o1 = new Override1();
              o1.display();
        }
  }
```

CODERINDEED

Static & Dynamic Binding

Compile time Binding

- At compile time, the call to a overridden method is resolved on the basis of reference variable.

- This is known as Compile time binding. But actual invocation of non-static method is done on the basis of Object, not on the basis of Reference Variable.

Dynamic Method Dispatch

- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

- It means that the non-static methods are invoked depending on the Object not on the reference Variable.

- Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

```
class Parent {
                    public void demo() {System.out.println("Parent Demo");}
                }

class Child extends Parent {
  public void demo() {System.out.println("Overridden Child Demo");}
  public void test() {System.out.println("Test");}
  }

class BindingDemo {
    public static void main(String rk[])  {
          Parent p = new Child();
          p.demo();          //At compile time it will check for demo() in Parent but it
                                 will execute demo() of child.
          p.test();          //Compile time error as test() is not available in Parent.

}}
```

Brainstorming Questions

CODERINDEED

Question1

What will be Output?

```
class Parent {
    public void drive() {System.out.print("Parent");}
    public int drive(int a) {System.out.print("Parent with Int");
    return 0;      }
}
class Child extends Parent {
    public void drive() {System.out.print("Child");    }
    public static void main(String … rk) {
    Parent p = new Child();      p.drive();
    }
}
```

Question2

What will be Output?

```
class Parent {
    public void drive() {System.out.print("Parent");}
    public int drive(int a) {System.out.print("Parent with Int");
    return 0;      }
}
class Child extends Parent {
    public void drive() {System.out.print("Child");     }
    public static void main(String … rk) {
    Parent p = new Child();      p.drive(10);
    }
}
```

CODERINDEED

Question3

What will be Output?

```
class Parent {
    public void drive() {System.out.print("Parent");}                    }

class Child extends Parent {
    public void drive() {System.out.print("Child");          }
    public int drive(int a) {System.out.print("Its Drive");
    return 0;       }

    public static void main(String … rk) {
    Parent p = new Child();         p.drive(10);
    }
}
```

Question4

What will be Output?

```
class Parent {
    public void drive() {System.out.print("Parent");}
    public int drive(int a) {System.out.print("Parent with Int");
    return 0;      }
}
class Child extends Parent {
    public void drive() {System.out.print("Child");  }
    public static void main(String … rk) {
    Child p = new Parent();      p.drive();
    }
}
```

CODERINDEED

Question5

What will be Output?

```
class Parent {
    public void drive() {System.out.print("Parent");}
    public int drive(int a) {System.out.print("Parent with Int");
    return 0;      }
}
class Child extends Parent {
    public int drive() {System.out.print("Child");  return 0;   }
    public static void main(String … rk) {
    Parent p = new Parent();    p.drive();
    }
}
```

CODERINDEED

Question1

What will be Output?

```
class Parent {
    public void drive(double x, int y) {System.out.print("First");  }

}
class Child extends Parent {
     public void drive(int x, double y) {System.out.print("Second");}
    public static void main(String...rk) {
    Parent p = new Parent();     p.drive(3, 5);
    }
}
```

CODERINDEED

*_Question2

What will be Output?

```
class Parent {
    public void drive(double x, int y) {System.out.print("First");  }
}
class Child extends Parent {
    public void drive(int x, double y) {System.out.print("Second");}
    public static void main(String...rk) {
            Child p = new Child();          p.drive(3, 5);
    }
}
```

RK_Question3***

CODERINDEED

What will be Output?

```
class Parent {
    public void drive(double x, int y) {System.out.print(“First");  }
    }
class Child extends Parent {
    public void drive(int x, double y) {System.out.print(“Second");}
    public static void main(String...rk) {
    Parent p = new Child();      p.drive(3, 5);
    }
}
```

CODERINDEED

Programming in Java

Methods and Constructors

By

coderindeed.in

METHODS

Method

- A method is a construct for grouping statements together to perform a function.

- A method that returns a value is called a *value retuning method*, and the method that does not return a value is called *void method*.

- In some other languages, methods are referred to as *procedures* or *functions*.
- A method which does not return any value is called a procedure.
- A method which returns some value is called a function.

Defining a Method

CODERINDEED

Syntax:

```
modifier returnType methodName (list of parameters)
   {
        // Body of the method(set of statements);
   }
```

Method Signature

Example:

```
public static void main (String args[])
        {
                …
        }
```

Method Header

Method Body

- Method header specifies the modifier, return type, method name and parameters of method.

```
public void display()
  {…}
```

- The variables defined in method header are called formal parameters or parameters.

```
int display(int x, int y)
  {…}
```

- When a method is invoked, a value as a parameter is passed which is known as actual parameters or arguments.

```
a.display (3, 5);
```

- Method body contains a set of statements that define the function to be performed by the method.

- A return statement using the keyword *return* is required for a value-returning method to return a result.

CODERINDEED

Calling a Method

- To use a method, we need to *call* or *invoke* it.
- There are two ways to call a method:
  - If the method returns a value, a call to method is usually treated as a value.

```
int  area = rectangleArea (4,6);
System.out.println( rectangleArea (4,6) );
```

  - If the method returns void, a call to method must be a statement.
  - For example, println method returns void

```
System.out.println("Hello…");
```

Exercise

Write a program to create a class BankAccount having instance variable *balance*.

Implement a method deposit(int amt) which receives the amount to be deposited as an argument and adds to the current balance.

Implement another method int withdraw() which asks the user to enter the amount to be withdrawn and updates the balance if having sufficient balance and return the new balance.

Invoke both the methods from TestBankAccount class.

CONSTRUCTOR

Constructors

- A **constructor** is a special method that is used to **initialize a newly created object.**
- It is called just after the memory is allocated for the object.
- It can be used to initialize the objects, to **required, or default values** at the time of object creation.
- Constructor cannot return values.
- Constructor has the same name as the class name.
- It is **not mandatory** for the coder to write constructor for the class.

Default Constructor

- If no user defined constructor is provided for a class, compiler initializes member variables to its default values.
  - numeric data types are set to 0
  - char data types are set to null character(‘’)
  - reference variables are set to null
  - boolean are set to false

- In order to create a Constructor observe the following rules:
  - It has the **same name** as the class
  - It should not return a value, not even ***void***

Defining a Constructor

- Like any other method

```
public class ClassName {

    // Data Fields…

    // Constructor
     public ClassName()
     {
          // Method Body Statements initialising Data Fields
     }

    //Methods to manipulate data fields
}
```

- Invoking:
  - There is NO explicit invocation statement needed: When the object creation statement is executed, the constructor method will be executed automatically.

Constructors

- **Constructor name is class name**. A constructors must have the *same name as the class its in*.

- **Default constructor**. If you don't define a constructor for a class, a *default (parameter-less) constructor* is automatically created by the compiler.

- The default constructor initializes all instance variables to default value (zero for numeric types, null for object references, and false for booleans).

Key Points

- Default constructor is created only if there are no constructors.
- If you define *any constructor* for your class, no default constructor is automatically created.
- There is *no return type* given in a constructor signature (header).
- There is *no return statement* in the body of the constructor.

Key Points

- The *first line* of a constructor must either be a call on another constructor in the same class (using this), or a call on the super-class constructor (using super).

- If the first line is neither of these, the compiler automatically inserts a call to the parameter-less super class constructor.

CODERINDEED

Parameterized Constructors

```
Cube1() { length = 10;
          breadth = 10;
          height = 10; }

Cube1(int l, int b, int h) { length = l;
                             breadth = b;
                             height = h;  }

public static void main(String[] args) {
                    Cube1 c1 = new Cube1();
                    Cube1 c2 = new Cube1(10, 20, 30);
   System.out.println("Volume of Cube1 is : " + c1.getVolume());
   System.out.println("Volume of Cube1 is : " + c2.getVolume());
                    }
    }
```

CODERINDEED

Constructor Overloading

```
class Cube1 {
int length, breadth, height;
 Cube1() { length = 10; breadth = 10; height = 10; }
 Cube1(int l) { length = l; }
 Cube1(int l, int b, int h) { length = l; breadth = b; height = h  }

 public static void main(String[] args) {
                        Cube1 c1 = new Cube1();
                        Cube1 c2 = new Cube1(50);
                        Cube1 c3 = new Cube1(10, 20, 30);

                        }
    }
```

CODERINDEED

Let's Do Some thing

Write a program to create a class named Patient which contains:

a. Attributes patient _name, age, contact_number
b. Constructor to initialize all the patient attributes
c. A method to display all the details of any patient

Create a Patient object and display all the details of that patient.

Future Work:

a. Assign unique patient_id to each patient automatically.
b. Store all the patient objects in an array or ArrayList.
c. Write a method to diplay all the details of a patient with a given patient_id.

Singleton Design Pattern

- Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the java virtual machine.

- The singleton class must provide a global access point to get the instance of the class.

Advantage:

Saves memory because object is not created at each request. Only single instance is reused again and again.

Usage of Singleton Design Pattern

- Singleton pattern is mostly used in multi-threaded and database applications.

- It is used in logging, caching, thread pools, configuration settings etc.

Creating Singleton Design Pattern

To create the singleton class, we need to have static member of class, private constructor and static factory method.

- Static member: It gets memory only once because of static, it contains the instance of the Singleton class.

- Private constructor: It will prevent to instantiate the Singleton class from outside the class.

- Static factory method: This provides the global point of access to the Singleton object and returns the instance to the caller.

Example

```
class MySingleton
{
   private static MySingleton ob;
   private MySingleton(){}
   public static MySingleton getInstance()
      {
    if (ob == null){
          ob = new MySingleton();
    return ob;
      }
}
```

CODERINDEED

Programming in Java

this, super, static & final Keywords

By

coderindeed.in

Contents

- final keyword
- this keyword
- static keyword
- super keyword

‘final’ Keyword

- ‘final’ keyword is used to:
  - declare variables that can be assigned value only once (constant).

```
final type identifier = expression;
```

  - prevent overriding of a method.

```
final return_type methodName (arguments if any)
{
    body;
}
```

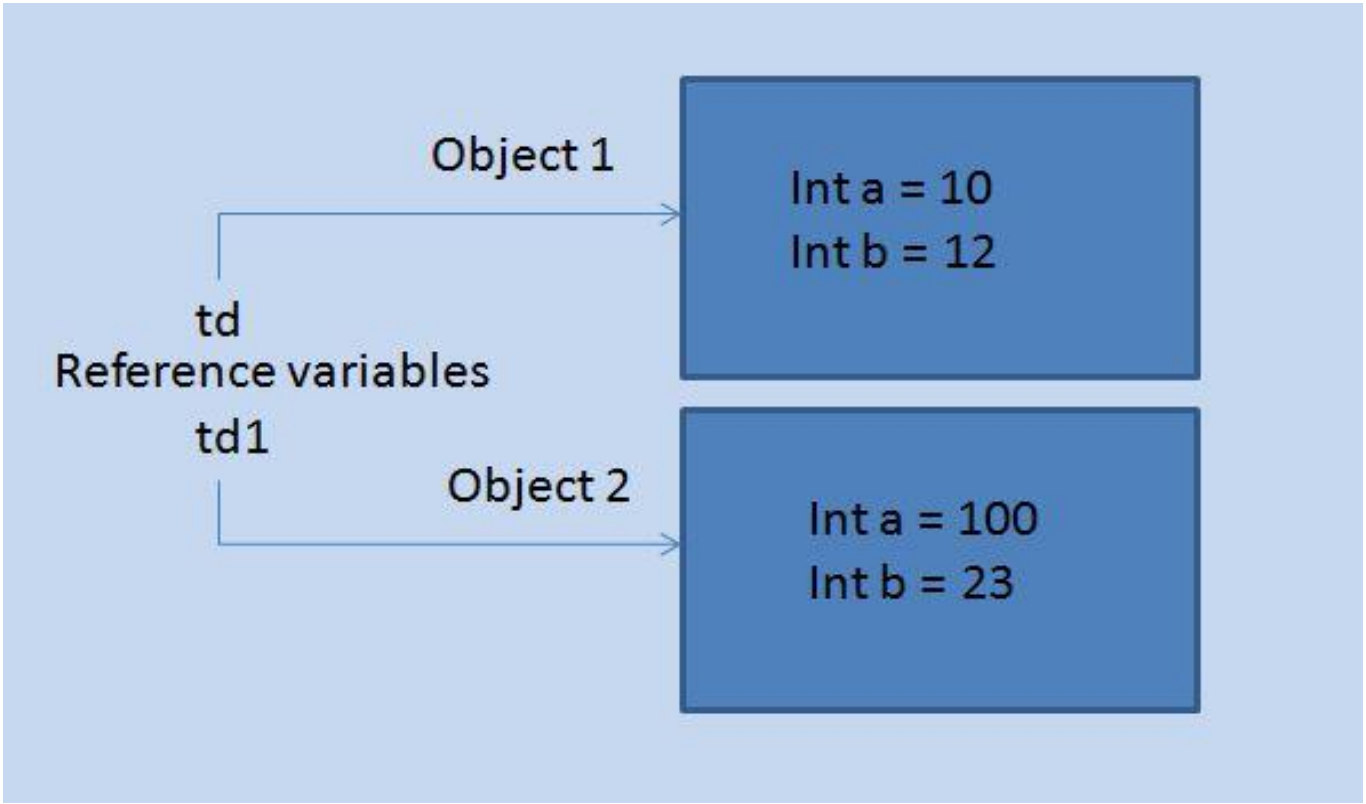
  - prevent inheritance from a class.

```
final class Class_Name
{
  class body;
}
```

CODERINDEED

‘this’ Keyword

- 'this' is used for pointing the current class instance.
- Within an instance method or a constructor, this is a reference to the *current object* — the object whose method or constructor is being called.



```
class ThisDemo1{
                int a = 0;
                int b = 0;
                ThisDemo1(int x, int y)
                    {
                       this.a = x;
                       this.b = y;
                     }

                public static void main(String [] args)
                     {
                        ThisDemo1 td = new ThisDemo1(10,12);
                        ThisDemo1 td1 = new ThisDemo1(100,23);
                        System.out.println(td.a);
                        System.out.println(td.B);
                        System.out.println(td1.a);
                       System.out.println(td1.B);
                    }
            }
```

Chaining of constructors using this keyword

- Chaining of constructor means calling one constructor from other constructor.
- We can invoke the constructor of same class using ‘this’ keyword.
- We can invoke the super class constructor from subclass constructor using ‘super’ keyword.

```
class ThisDemo{
        public ThisDemo() {
                this(10);   System.out.println("First Constructor");
                }
        public ThisDemo(int a) // overloaded constructor
                {
                this(10,20);   System.out.println("Second Constructor");
                }
        public ThisDemo( int a, int B) // another overloaded constructor
                {
                this(“Ravi Kant");   System.out.println("Third Constructor");
                }
        public ThisDemo(String s) // and still another
                {
                System.out.println("Fourth Constructor");
                }
        public static void main(String args[]) {
                        ThisDemo first = new ThisDemo();
                }
    }
```

Constructor Chaining using ‘super’

```
class Parent{
        public Parent() {  System.out.println(“Parent Class");
                          }
        }

class Parent2 extends Parent {
        public Parent2(int x) {    super();
                                    System.out.println(“Parent2 Class: ”+ x );
                                }
        }
class Child extends Parent2{
                public Child() {          System.out.println(“Child Class");
                                  }
                public static void main(String args[]) {  Child c = new Child();
                                                        }
            }
```

‘static’ Keyword

- used to represent class members
- Variables can be declared with the “static” keyword.

static int y = 0;

- When a variable is declared with the keyword “static”, its called a “**class variable**”.
- All instances share the same copy of the variable.
- A class variable can be accessed directly with the class, without the need to create a instance.

- Note: There's no such thing as static classs. “static” in front of class creates compilation error.

CODERINDEED

static methods

- Methods can also be declared with the keyword “static”.
- When a method is declared static, it can be used without creating an object.

```
class T2 {
            static int triple (int n)
                  {return 3*n;}
          }

class T1 {
             public static void main(String[] arg)
                  {
                     System.out.println( T2.triple(4) );
                     T2 x1 = new T2();
                     System.out.println( x1.triple(5) );
                  }
           }
```

- Methods declared with “static” keyword are called “**class methods**”.
- Otherwise they are “**instance methods**”.
- **Static Methods Cannot Access Non-Static Variables.**
- The following gives a compilation error, unless x is also static.

```
class T2 {
        int x = 3;
        static int returnIt () { return x;}
    }

class T1 {
        public static void main(String[] arg) {
        System.out.println( T2.returnIt() ); }
    }
```

CODERINDEED

‘super’ Keyword

- ‘super’ keyword is used to:
  - invoke the super-class constructor from the constructor of a sub-class.

    super (arguments if any);

  - invoke the method of super-class on current object from sub-class.

    super.methodName (arguments if any);

  - refer the super-class data member in case of name-conflict between super and sub-class data members.

    super.memberName;

Important

- ‘this’ and ‘super’ both are non-static.
- So they can not be used inside static scope.

CODERINDEED

Programming in Java

Abstract class and Interface

By

coderindeed.in

Contents

- Abstract Class
- Object Class

Abstract Class

- An *abstract class* is a class that is declared abstract.
- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:

```
abstract void moveTo(double deltaX, double deltaY);
```

- Abstract class may or may not include abstract methods.
- Abstract classes cannot be instantiated, but they can be sub-classed.

- If a class includes abstract methods, the class itself *must* be declared abstract, as in:

```
public abstract class GraphicObject
        {       // declare fields
                // declare non-abstract methods
                abstract void draw();
        }
```

NOTE:

- When an abstract class is sub-classed, the subclass usually provides implementations for all of the abstract methods in its parent class.
- However, if it does not, the subclass must also be declared abstract.

- It can be considered as a blank form consisting of several sections, some of them may be optional (non-abstract method) and some may be mandatory (abstract)…

- Can we Override a constructor?

  If yes, How? if No, Why?

INTERFACES

Interfaces

- An interface is a collection of abstract methods.
- An interface is not a class.
- A class describes the attributes and behaviors of an object whereas an interface *contains behaviors* that a class implements.
- A class implements an interface, thereby inheriting the abstract methods of the interface.
- Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

Properties of Interfaces

- The **interface** keyword is used to declare an interface.
- Interfaces have the following properties:
  - An interface is implicitly abstract. We do not need to use the **abstract** keyword when declaring an interface.
  - Each method in an interface is also implicitly public and abstract, so the abstract keyword is not needed.
  - Each variable in an interface is implicitly public, static and final.

Interface Vs Class

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte-code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding byte-code file must be in a directory structure that matches the package name.

Interface Vs Class

An interface is different from a class in several ways, including:

- We cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Implementing Interfaces

- When a class implements an interface, then it has to perform the specific behaviors of the interface.
- If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
- A class uses the **implements** keyword to implement an interface.
- The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
interface Animal
        {
                void eat();
               public void travel();
        }
```

Example

```
public class Mammal implements Animal
    {
            public void eat()
                    {
                            System.out.println("Mammal eats");
                    }
            public void travel()
                    {
                            System.out.println("Mammal travels");
                    }
            public int noOfLegs()
                    {               return 0; }
            public static void main(String args[])
                    {
                            Mammal m = new Mammal();
                            m.eat();
                            m.travel();
                    }
     }
```

Abstract Class Vs Interfaces

Abstract Class	Interface
May contain non-abstract methods and non-static non final data members	Contains only method declaration and static final data members
Multiple Inheritance is not supported through classes	Multiple Inheritance through Interfaces is supported
Classes provide static classing environment	Interfaces provide dynamic classing environment
Inheritance using ‘extends’ keyword	Inheritance using ‘implements’ keyword

Object Class

- There is one special class, called Object, defined by Java.
- All other classes are subclasses of Object.
- A reference variable of type Object can refer to an object of any other class.
- Arrays are implemented as classes, so a variable of type Object can also refer to any array.
- In Object class, getClass( ), notify( ), notifyAll( ), and wait( ) are declared as final.

Methods in Object class

Method	Purpose
Object clone( )	Creates a new object that is the same as the object being cloned.
boolean equals(Object *object*)	Determines whether one object is equal to another.
void finalize( )	Called before an unused object is recycled.
Class getClass( )	Obtains the class of an object at run time.
int hashCode( )	Returns the hash code associated with the invoking object.
void notify( )	Resumes execution of a thread waiting on the invoking object.
void notifyAll( )	Resumes execution of all threads waiting on the invoking object.
String toString( )	Returns a string that describes the object.
void wait( ) void wait(long *milliseconds*) void wait(long *milliseconds*, int *nanoseconds*)	Waits on another thread of execution.

CODERINDEED

Programming in Java

String Handling

By
coderindeed.in
Asst. Professor

Lovely Professional University, Punjab

Introduction

- Every string we create is actually an object of type String.
- String constants are actually String objects.

- Example:

```
System.out.println("This is a String, too");
```

- Objects of type String are immutable i.e. once a String object is created, its contents cannot be altered.

Why String is Immutable or Final?

- String has been widely used as parameter for many java classes e.g. for opening network connection we can pass hostname and port number as string ,
- we can pass database URL as string for opening database connection,
- we can open any file in Java by passing name of file as argument to File I/O classes.

- In case if String is not immutable , this would lead serious security threat, means some one can access to any file for which he has authorization and then can change the file name either deliberately or accidentally and gain access of those file.

Introduction

- In java, four predefined classes are provided that either represent strings or provide functionality to manipulate them. Those classes are:
  - String
  - StringBuffer
  - StringBuilder
  - *StringTokenizer*

- String, StringBuffer, and StringBuilder classes are defined in java.lang package and all are final.

- All of them implement the CharSequence interface.

Important

- String Objects are IMMUTABLE.
- StringBuffer is MUTABLE and SYNCHRONIZED.
- StringBuilder is MUTABLE and NON-SYNCHRONIZED.

Why String Handling?

String handling is required to perform following operations on some string:

- compare two strings
- search for a substring
- concatenate two strings
- change the case of letters within a string

Creating String objects

```
class StringDemo
        {
          public static void main(String args[])
                {
                  String strOb1 = “Ravi";
                  String strOb2 = “LPU";
                  String strOb3 = strOb1 + " and " + strOb2;
                  System.out.println(strOb1);
                  System.out.println(strOb2);
                  System.out.println(strOb3);
                }
        }
```

String Class

String Constructor:

public String ()
public String (String)
public String (char [])
public String (byte [])
public String (char [], int offset, int no_of_chars)
public String (byte [], int offset, int no_of _bytes)

Examples

```
char [] a = {'c', 'o', 'n', 'g', 'r', 'a', 't', 's'};
byte [] b = {82, 65, 86, 73, 75, 65, 78, 84};

String s1 = new String (a); System.out.println(s1);

String s2 = new String (a, 1,5); System.out.println(s2);

String s3 = new String (s1); System.out.println(s3);

String s4 = new String (b); System.out.println(s4);

String s5 = new String (b, 4, 4); System.out.println(s5);
```

String Concatenation

- Concatenating Strings:

```
String age = "9";
String s = "He is " + age + " years old.";
System.out.println(s);
```

- Using concatenation to prevent long lines:

```
String longStr = “This could have been” +
                 “a very long line that would have” +
                 “wrapped around.  But string”+
                “concatenation  prevents this.”;
System.out.println(longStr);
```

String Concatenation with Other Data Types

- We can concatenate strings with other types of data.

Example:

```
int age = 9;
String s = "He is " + age + " years old.";
System.out.println(s);
```

Methods of String class

- String Length:

  length() returns the length of the string i.e. number of characters.

public int length()

Example:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

concat( ): used to concatenate two strings.

String concat(String str)

- This method creates a new object that contains the invoking string with the contents of str appended to the end.

- concat( ) performs the same function as +.

Example:

```
String s1 = "one"; String s2 = s1.concat("two");
```

- It generates the same result as the following sequence:

```
String s1 = "one"; String s2 = s1 + "two";
```

Character Extraction

- charAt(): used to obtain the character from the specified index from a string.

public char charAt (int index);

Example:

```
char ch;
ch = "abc".charAt(1);
```

Methods Cont…

- toCharArray(): returns a character array initialized by the contents of the string.

public char [] toCharArray();

Example:

```
String s = "India";
char c[] = s.toCharArray();
    for (int i=0; i<c.length; i++)
        {
          if (c[i]>= 65 && c[i]<=90)
                c[i] += 32;
                System.out.print(c[i]);
        }
```

Methods Cont…

- getChars(): used to obtain set of characters from the string.

public void getChars(int start_index, int end_index, char[], int offset)

Example:
```
String s = "KAMAL";
char b[] = new char [10];
b[0] = 'N';    b[1] = 'E';
b[2] = 'E';    b[3] = 'L';
s.getChars(0, 5, b, 4);
System.out.println(b);
```

String Comparison

- equals(): used to compare two strings for equality.
  Comparison is case-sensitive.

  public boolean equals (Object str)

- equalsIgnoreCase( ): To perform a comparison that ignores case differences.

Note:

- This method is defined in Object class and overridden in String class.
- equals(), in Object class, compares the value of reference not the content.
- In String class, equals method is overridden for content-wise comparison of two strings.

Example

```
class equalsDemo {
        public static void main(String args[]) {
                String s1 = "Hello";
                String s2 = "Hello";
                String s3 = "Good-bye";
                String s4 = "HELLO";
                System.out.println(s1 + " equals " + s2 + " -> " +
                s1.equals(s2));
                System.out.println(s1 + " equals " + s3 + " -> " +
                s1.equals(s3));
                System.out.println(s1 + " equals " + s4 + " -> " +
                s1.equals(s4));
                System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> “
                                        +s1.equalsIgnoreCase(s4));
                }
        }
```

String Comparison

- startsWith( ) and endsWith( ):
  - The startsWith( ) method determines whether a given String begins with a specified string.
  - Conversely, endsWith( ) determines whether the String in question ends with a specified string.

boolean startsWith(String str)

boolean endsWith(String str)

String Comparison

compareTo( ):

- A string is less than another if it comes before the other in dictionary order.
- A string is greater than another if it comes after the other in dictionary order.

int compareTo(String str)

Value	Meaning
Less than zero	The invoking string is less than *str*.
Greater than zero	The invoking string is greater than *str*.
Zero	The two strings are equal.

Example

```
class SortString {
    static String arr[] = {"Now", "is", "the", "time", "for", "all", "good", "men",
                           "to", "come", "to", "the", "aid", "of", "their", "country"};
    public static void main(String args[]) {
            for(int j = 0; j < arr.length; j++) {
                    for(int i = j + 1; i < arr.length; i++) {
                            if(arr[i].compareTo(arr[j]) < 0) {
                                    String t = arr[j];
                                    arr[j] = arr[i];
                                    arr[i] = t;
                            }
                    }
              System.out.println(arr[j]);
            }
        }
   }
```

Searching Strings

- The String class provides two methods that allow us to search a string for a specified character or substring:

  indexOf( ): Searches for the first occurrence of a character or substring.

  int indexOf(int ch)

  lastIndexOf( ): Searches for the last occurrence of a character or substring.

  int lastIndexOf(int ch)

- To search for the first or last occurrence of a substring, use

  int indexOf(String str)

  int lastIndexOf(String str)

- We can specify a starting point for the search using these forms:

  int indexOf(int ch, int startIndex)
  int lastIndexOf(int ch, int startIndex)
  int indexOf(String str, int startIndex)
  int lastIndexOf(String str, int startIndex)

- Here, startIndex specifies the index at which point the search begins.

- For indexOf( ), the search runs from startIndex to the end of the string.

- For lastIndexOf( ), the search runs from startIndex to zero.

Example

```
class indexOfDemo {
    public static void main(String args[]) {
            String s = "Now is the time for all good men " +
                        "to come to the aid of their country.";
            System.out.println(s);
            System.out.println("indexOf(t) = " + s.indexOf('t'));
            System.out.println("lastIndexOf(t) = " + s.lastIndexOf('t'));
            System.out.println("indexOf(the) = " + s.indexOf("the"));
            System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
            System.out.println("indexOf(t, 10) = " + s.indexOf('t', 10));
            System.out.println("lastIndexOf(t, 60) = " + s.lastIndexOf('t', 60));
            System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
            System.out.println("lastIndexOf(the, 60) = " + s.lastIndexOf("the", 60));
            }
    }
```

Modifying a String

- Because String objects are immutable, whenever we want to modify a String, it will construct a new copy of the string with modifications.

- substring(): used to extract a part of a string.

public String substring (int start_index)

public String substring (int start_index, int end_index)

Example: String s = “ABCDEFG”;

String t = s.substring(2); System.out.println (t);

String u = s.substring (1, 4); System.out.println (u);

Note: Substring from start_index to end_index-1 will be returned.

replace( ): The replace( ) method has two forms.

- The first replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace(char original, char replacement)

- Here, original specifies the character to be replaced by the character specified by replacement.

Example: String s = "Hello".replace('l', 'w');

- The second form of replace( ) replaces one character sequence with another. It has this general form:

String replace(CharSequence original, CharSequence replacement)

Let's Do Something…

Write a program which prompts the user to enter a paragraph in Present continuous tense and display it in Past Continuous.

trim( )

- The trim( ) method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

String trim( )

Example:

```
String s = "   Hello World    ".trim();
```

This puts the string “Hello World” into s.

Changing the Case of Characters Within a String

toLowerCase() & toUpperCase()

- Both methods return a String object that contains the uppercase or lowercase equivalent of the invoking String.

String toLowerCase( )
String toUpperCase( )

Split()

public String [ ] split (String regex)

public String [ ] split (String regex, int limit)

Returns: An array of strings computed by splitting the given string.

Throws: PatternSyntaxException - if the provided regular expression's syntax is invalid.

Example:

```
String str = "have-fun-in-java@blogspot@in";
   String [] s = str.split("-", -2);
   for(String  x: s) System.out.println(x);
```

Limit parameter can have 3 values:

limit > 0 : If this is the case then the pattern will be applied at most limit-1 times, the resulting array's length will not be more than n, and the resulting array's last entry will contain all input beyond the last matched pattern.

limit < 0 : In this case, the pattern will be applied as many times as possible, and the resulting array can be of any size.

limit = 0 : In this case, the pattern will be applied as many times as possible, the resulting array can be of any size, and trailing empty strings will be discarded.

join()

public static String join(CharSequence delimiter, CharSequence... elements)

Returns a new String composed of copies of the CharSequence elements joined together with a copy of the specified delimiter. *(Introduced in jdk 8)*

Example:

```
String joinString1=String.join(“-”, “have",“fun",“in“, “java”);
System.out.println(joinString1);
```

CODERINDEED

Programming in Java

Topic: StringBuilder Class

By
coderindeed.in

Introduction

- Java StringBuilder class is used to create mutable (modifiable) string.
- The Java StringBuilder class is same as StringBuilder class except that it is non-synchronized.
- It is available since JDK 1.5.

StringBuilder Constructors

- StringBuilder defines these four constructors:
  - StringBuilder( )
  - StringBuilder(int cap)
  - StringBuilder(String str)
- The default constructor reserves room for 16 characters without reallocation.
- By allocating room for a few extra characters(size +16), StringBuilder reduces the number of reallocations that take place.

Brain Storming

```
StringBuilder sb = new StringBuilder();
System.out.println(sb.capacity());

StringBuilder sb = new StringBuilder(65);
System.out.println(sb.capacity());

StringBuilder sb = new StringBuilder(“A”);
System.out.println(sb.capacity());

StringBuilder sb = new StringBuilder('A');
System.out.println(sb.capacity());
```

StringBuilder Methods

length( ) and capacity( )

The current length of a StringBuilder can be found via the length( ) method, while the total allocated capacity can be found through the capacity( ) method.

int length( )

int capacity( )

Example:

```
class StringBuilderDemo {
        public static void main(String args[]) {
                StringBuilder sb = new StringBuilder(“New Zealand");
                System.out.println("length = " + sb.length());
                System.out.println("capacity = " + sb.capacity());
            }
        }
```

ensureCapacity( )

- If we want to preallocate room for a certain number of characters after a StringBuilder has been constructed, we can use ensureCapacity( ) to set the size of the buffer.

- This is useful if we know in advance that we will be appending a large number of small strings to a StringBuilder.

void ensureCapacity(int capacity)

- Here, capacity specifies the size of the buffer.

Important

- When the length of StringBuilder becomes larger than the capacity then memory reallocation is done:

- In case of StringBuilder, reallocation of memory is done using the following rule:

- If the new_length <= 2*(original_capacity + 1), then
  new_capacity = 2*(original_capacity + 1)

- Else, new_capacity = new_length.

Brainstorming

What will be the output?

```
class DemoString{
   public static void main(String...rk)     {
    StringBuilder b = new StringBuilder("abcdef");
    //b.delete(4,6);
    b.ensureCapacity(22);
    System.out.print(b.capacity());
    b.ensureCapacity(23);
    System.out.print(b.capacity());
     }
}
```

setLength( )

- used to set the length of the buffer within a StringBuilder object.

void setLength(int length)

- Here, length specifies the length of the buffer.
- When we increase the size of the buffer, null characters are added to the end of the existing buffer.
- If we call setLength( ) with a value less than the current value returned by length( ), then the characters stored beyond the new length will be lost.

charAt( ) and setCharAt( )

- The value of a single character can be obtained from a StringBuilder via the charAt( ) method.

- We can set the value of a character within a StringBuilder using setCharAt( ).

char charAt(int index)

void setCharAt(int index, char ch)

getChars( )

- getChars( ) method is used to copy a substring of a StringBuilder into an array.

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart)

append( )

- The append( ) method concatenates the string representation of any other type of data to the end of the invoking StringBuilder object.

- It has several overloaded versions.
  - StringBuilder append(String str)
  - StringBuilder append(int num)
  - StringBuilder append(Object obj)

Example

```
class appendDemo {
        public static void main(String args[]) {
                String s;
                int a = 42;
                StringBuilder sb = new StringBuilder(40);
                s = sb.append("a = ").append(a).append("!")
                    .toString();
                System.out.println(s);
                }
        }
```

insert( )

- The insert( ) method inserts one string into another.
- It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.
- This string is then inserted into the invoking StringBuilder object.
    - StringBuilder insert(int index, String str)
    - StringBuilder insert(int index, char ch)
    - StringBuilder insert(int index, Object obj)

Example

```
class insertDemo {
        public static void main(String args[]) {
                StringBuilder sb = new StringBuilder("I Java!");
                sb.insert(2, “don’t like ");
                System.out.println(sb);
               }
        }
```

reverse( )

- Used to reverse the characters within a StringBuilder object.
- This method returns the reversed object on which it was called.

StringBuilder reverse()

Example:

```
class ReverseDemo {
        public static void main(String args[]) {
        StringBuilder s = new StringBuilder(“Banana");
        System.out.println(s);
        s.reverse();
        System.out.println(s);
        }
}
```

Let's Do Something…

Write a program which prompts the user to enter a String.

- Check Whether the String is Palindrome or Not?

delete( ) and deleteCharAt( )

- Used to delete characters within a StringBuilder.

  StringBuilder delete(int startIndex, int endIndex)
  StringBuilder deleteCharAt(int index)

- The delete( ) method deletes a sequence of characters from the invoking object (from startIndex to endIndex-1).

- The deleteCharAt( ) method deletes the character at the specified index.

- It returns the resulting StringBuilder object.

Example

```
class deleteDemo {
        public static void main(String args[]) {
        StringBuilder sb = new StringBuilder(“She is not a good
   girl.”);
        sb.delete(7, 11);
        System.out.println("After delete: " + sb);
        sb.deleteCharAt(7);
        System.out.println("After deleteCharAt: " + sb);
        }
   }
```

replace( )

- Used to replace one set of characters with another set inside a StringBuilder object.

StringBuilder replace(int startIndex, int endIndex, String str)

- The substring being replaced is specified by the indexes startIndex and endIndex.
- Thus, the substring at startIndex through endIndex–1 is replaced.
- The replacement string is passed in str.
- The resulting StringBuilder object is returned.

Example

```
class replaceDemo {
        public static void main(String args[]) {
            StringBuilder sb = new StringBuilder("This is a
  test.");
            sb.replace(5, 7, "was");
            System.out.println("After replace: " + sb);
        }
  }
```

substring( )

- Used to obtain a portion of a StringBuilder by calling substring( ).

String substring(int startIndex)
String substring(int startIndex, int endIndex)

- The first form returns the substring that starts at startIndex and runs to the end of the invoking StringBuilder object.

- The second form returns the substring that starts at startIndex and runs through endIndex–1.

CODERINDEED

Programming in Java

Topic: Date Time API

By

coderindeed.in

Contents…

Introduction

- New DateTime API is introduced in jdk8.
- LocalDate, LocalTime and LocalDateTime classes are provided in java.time package.

Java Date and Time API goals

- Classes and methods should be straight forward.
- Instances of Date and Time objects should be immutable.
- Should be thread safe.
- Use ISO standard to define Date and Time.

Working with Local Date and Time

- Java.time package provides two classes for working with local Date and Time.
- LocalDate
  - Does not include time
  - A year-month-day representation
  - toString – ISO 8601 format(YYYY-MM-DD)

- LocalTime
  - Does not include date
  - Stores hours:minutes:seconds:nanoseconds
  - toString- (HH:mm:ss.SSS)

LocalDate, LocalTime and LocalDateTime

- They are local in the sense that they represent date and time from the context of one observer, in contrast to time zones.
- All the core classes in the new API are constructed by factory methods.
- When constructing a value through its fields, the factory is called *of*.
- There are also parse methods that take strings as parameters.

LocalDate Class

LocalDate Class

- A date without a time-zone in the ISO-8601 calendar system, such as 2007-12-03.
- LocalDate is an immutable date-time object that represents a date, often viewed as year-month-day.
- Other date fields, such as day-of-year, day-of-week and week-of-year, can also be accessed.
- This class does not store or represent a time or time-zone so its portable across time zones.

Methods of LocalDate

- public static LocalDate now()
- public static LocalDate now(ZoneId zone)

- public static LocalDate of(int year, Month month, int dayOfMonth)

- public static LocalDate of(int year, int month, int dayOfMonth)

Note: *DateTimeException can be thrown.*

- public static LocalDate parse(CharSequence text)

Note: *DateTimeParseException can be thrown.*

Example

```
LocalDate ldt = LocalDate.now();

ldt = LocalDate.of(2015, Month.FEBRUARY, 28);

ldt = LocalDate.of(2015, 2, 13);

ldt = LocalDate.parse("2017-02-28”);
```

LocalTime Class

LocalTime Class

- A time without a time-zone in the ISO-8601 calendar system, such as 10:15:30. 13
- LocalTime is an immutable date-time object that represents a time, often viewed as hour-minute-second.
- Time is represented to nanosecond precision.
- For example, the value "13:45:30.123" can be stored in a LocalTime.
- This class does not store or represent a date or time-zone.

Methods of LocalTime

Methods

- public static LocalTime now()
- public static LocalTime now(ZoneId zone)

- public static LocalTime of(int hour, int minute)
- public static LocalTime of(int hour, int minute, int second)
- public static LocalTime of(int hour, int min, int sec, int nsec)

- public static LocalTime parse(CharSequence text)

LocalDateTime Class

LocalDateTime Class

- A date-time without a time-zone in the ISO-8601 calendar system, such as 2007-12-03T10:15:30.

- LocalDateTime is an immutable date-time object that represents a date-time, often viewed as year-month-day-hour-minute-second.
- Other date and time fields, such as day-of-year, day-of-week and week-of-year, can also be accessed.

- Time is represented to nanosecond precision.
- For example, the value "2nd October 2007 at 13:45.30.123456789" can be stored in a LocalDateTime.

Methods of LocalDateTime

Methods

- public static LocalDateTime now()
- public static LocalDateTime now(ZoneId zone)

- public static LocalDateTime of(int year, int mnth, int day, int hour, int mint)
- public static LocalDateTime of(int year, int mnth, int day, int hour, int mint, int sec)
- public static LocalDateTime of(int year, int mnth, int day, int hour, int mint, int sec, int nsec)
- public static LocalDateTime of(LocalDate d, LocalTime t)
- public static LocalDateTime parse(CharSequence text)

Working with dates and times across time zones

Date/Time Formatting

- java.time.format.DateTimeFormatter class is used for printing and parsing date-time objects.

- This class works by using:

  Predefined constants, such as ISO_LOCAL_DATE

  Pattern letters, such as yyyy-MMM-dd

  Localized styles, such as long or medium

Date/Time Formatting

- The date-time classes provide two methods - one for formatting and one for Parsing.

Formatting Example:

```
LocalDateTime dt =
LocalDateTime.of( 2010, Month.JULY, 03, 09, 0, 30 );
String isoDateTime =
dt.format(DateTimeFormatter.ISO_DATE_TIME);
```

Parsing Example:

```
LocalDate dt = LocalDate.parse("2014/09/19 14:05:12",
DateTimeFormatter.ofPattern( "yyyy/MM/dd kk:mm:ss" ) );
```

CODERINDEED

Programming in Java

Nested Classes

By

coderindeed.in

Contents

- Nested Class
- Static Nested Class
- Inner Class
- Local Class
- Anonymous Class

Nested Class

- The Java programming language allows us to define a class within another class. Such a class is called a *nested class*.

Example:

```
class OuterClass
        {
                ...
         class NestedClass
                {
                    ...
                }
         }
```

Types of Nested Classes

- A nested class is a member of its enclosing class.
- Nested classes are divided into two categories:
  - static
  - non-static
- Nested classes that are declared static are simply called static nested classes.
- Non-static nested classes are called inner classes.

CODERINDEED

Why Use Nested Classes?

- Logical grouping of classes—If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together.

- Increased encapsulation—Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.

- More readable, maintainable code—Nesting small classes within top-level classes places the code closer to where it is used.

Static Nested Classes

- A static nested class is associated with its outer class similar to class methods and variables.
- A static nested class cannot refer directly to instance variables or methods defined in its enclosing class.
- It can use them only through an object reference.
- Static nested classes are accessed using the enclosing class name:

  OuterClass.StaticNestedClass

- For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass  nestedObject =
                                    new  OuterClass.StaticNestedClass();
```

CODERINDEED

Inner Classes

- An inner class is associated with an instance of its enclosing class and has direct access to that object's methods and fields.

- Because an inner class is associated with an instance, it cannot define any static members itself.

- Objects that are instances of an inner class exist *within* an instance of the outer class.
- Consider the following classes:

```
class OuterClass {
        ...
        class InnerClass { ... }
}
```

- An instance of InnerClass can exist only within an instance of OuterClass and has direct access to the methods and fields of its enclosing instance.

- To instantiate an inner class, we must first instantiate the outer class. Then, create the inner object within the outer object.

- Syntax:

```
OuterClass.InnerClass  innerObject =
                              outerObject.new  InnerClass();
```

- Additionally, there are two special kinds of inner classes:
  - local classes and
  - anonymous classes (also called anonymous inner classes).

Let's Do Something…

- Write a program to create a class Person such that a person can have an aadhar card.
- If Aadhar Card is already created then return the same object other wise create the new Aadhar Object.
- Aadhar is a class and make sure that only person can have aadhar object.

Local Classes

- Local classes are *classes that are defined in a block*, which is a group of zero or more statements between balanced braces.
- For example, we can define a local class in a method body, a for loop, or an if clause.
- A local class has access to the members of its enclosing class.
- A local class has access to local variables. However, a local class can only access local variables that are final or effectively final.

Important

- A local class has access to local variables. However, a local class can only access local variables that are final.

- Starting in Java SE 8, a local class can access local variables and parameters of the enclosing block that are final or effectively final.

- A variable or parameter whose value is never changed after it is initialized is effectively final.

Anonymous Classes

- Anonymous classes enable us to declare and instantiate a class at the same time.

- They are like local classes except that they do not have a name.

- The anonymous class expression consists of the following:

  1. The new operator
  2. The name of an interface to implement or a class to extend.
  3. Parentheses that contain the arguments to a constructor, just like a normal class instance creation expression.
  4. A body, which is a class declaration body. More specifically, in the body, method declarations are allowed but statements are not.

- Anonymous classes have the same access to local variables of the enclosing scope as local classes:

- An anonymous class has access to the members of its enclosing class.
- An anonymous class cannot access local variables in its enclosing scope that are not declared as final.

- Anonymous classes also have the same restrictions as local classes with respect to their members:

- We cannot declare static initializers or member interfaces in an anonymous class.
- An anonymous class can have static members provided that they are constant variables.

- Note that we can declare the following in anonymous classes:

- Fields
- Extra methods (even if they do not implement any methods of the supertype)
- Local classes
- ***we cannot declare constructors in an anonymous class***.

Note:

When we compile a nested class, two different class files will be created with names

Outerclass.class

Outerclass$Nestedclass.class

- Local Class is named as Outerclass$1Localclass.class
- Anonymous class is named as Outerclass$1.class

CODERINDEED

Programming in Java

Topic: Interfaces & Lambda Expressions

By

coderindeed.in

Contents

- Interface
- Functional Interface
- Introduction of Lambda Expressions

Interface

- An interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types.
- Method bodies exist only for default methods and static methods.
- Interfaces cannot be instantiated, they can only be implemented by classes or extended by other interfaces.

Defining Interface

```
public interface My
   {
        int x = 10;                  // by default public static and final
        void demo();
                                     // by default public and abstract
        default void show()  {…}
                                     // default method, public
        static void test(){…}
                                     // static method, public
   }
```

Interface Methods

We can have three types of methods in an interface:

1. Abstract methods: Method without body and qualified with abstract keyword

2. static methods: Method with body and qualified with static keyword

3. default method: Method with body and qualified with default keyword

Default Methods

- Default Method allows to add new methods to the interfaces without breaking the existing implementation of an interface.

- It provides flexibility to allow interface define implementation which will be used as default in the situation where a concrete class fails to provide an implementation for that method.

- It can be overridden or made abstract in the inherited class or interface.

Implementing Interfaces

- When a class implements an interface, then it has to perform the specific behaviors of the interface.
- If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.
- A class uses the **implements** keyword to implement an interface.
- The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
interface Animal
        {
                void eat();
               public void travel();
        }
```

Example

```
public class Mammal implements Animal
    {
            public void eat()
                    {
                            System.out.println("Mammal eats");
                    }
            public void travel()
                    {
                            System.out.println("Mammal travels");
                    }
            public int noOfLegs()
                    {               return 0; }
            public static void main(String args[])
                    {
                            Mammal m = new Mammal();
                            m.eat();
                            m.travel();
                    }
     }
```

Abstract Class Vs Interfaces

Abstract Class	Interface
May contain non-abstract methods and non-static non final data members	May contain abstract methods, default methods and static methods.
Multiple Inheritance is not supported through classes	Multiple Inheritance through Interfaces is supported
Classes provide static classing environment	Interfaces provide dynamic classing environment
Inheritance using ‘extends’ keyword	Inheritance using ‘implements’ keyword

Functional Interface

- A functional interface is an interface that specifies only one abstract method.
- Also known as *Single Abstract Method (SAM)* interfaces.
- Introduced with jdk8.
- @FunctionalInterface annotation can be used for declaring a functional interface which results in an error when contract of Functional Interface is violated.

Example:

```
@FunctionalInterface
interface MyInterface
{
        int test(int n);
}
```

Note: If an interface declares an abstract method overriding one of the public methods of Object class, that does not count towards the interface's abstract method.

Example:

```
@FunctionalInterface
interface MyInterface
{
        int test(int n);
        public boolean equals(Object ob);
        public String toString();
}
```

Lambda Expression

- A lambda expression is, essentially, an anonymous (unnamed) method which is used to implement a method defined by a functional interface.
- Lambda expressions are also commonly referred to as closures.
- Lambda expressions are implementation of only abstract method of functional interface that is being implemented or instantiated anonymously.

Fundamentals of Lambda Expressions

- Lambda Expression introduces a new operator (->) which is referred to as the lambda operator or the arrow operator.
- It divides a lambda expression into two parts.
- The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.
- On the right side is the lambda body, which specifies the actions of the lambda expression.

Example

```
int sum(int a, int b)
   {
         return (a+b);
   }
```

Using Lambda:

```
(int a, int b)  ->  return(a+b);
```

OR

```
(a, b) -> return(a+b);
```

Important

- When a lambda expression has only one parameter, it is not necessary to surround the parameter name with parentheses when it is specified on the left side of the lambda operator.

Example:

```
boolean isEven(int n){
                        return (n%2==0);
                }
```

Using Lambda Expression: `n -> (n % 2)==0;`

Structure of Lambda Expressions

Argument List:

- A lambda expression can contain zero or more arguments.

```
// No argument
        ( ) -> { System.out.println("No argument"); }

// Single argument
        (int arg) -> { System.out.println("One argument : " + arg); }

// More than one arguments
        ( int arg1, String arg2 ) -> { System.out.println("Multiple
    Arguments:); }
```

Structure of Lambda Expressions

Argument List:

- We can eliminate the argument type while passing it to lambda expressions, those are inferred types. i.e. (int a) and (a) both are same.

  But we can not use inferred and declared types together

  ```
  ( int arg1, arg2 ) -> { … }          // This is invalid
  ```

- We can also eliminate "()" if there is only argument.

- More than one arguments are separated by comma (,) operator.

Structure of Lambda Expressions

Body of a lambda expression:

- Body can be a single expression or a statement block.
- When the body consist of a single expression, it is called as Expression Lambda or Expression Body.
- When the code on the right side of the lambda operator consists of a block of code that can contain more than one statement, It is called as block body or Block Lambda.

Characteristics of Lambda Expressions

- Optional type declaration
- Optional parenthesis around parameter
- Optional curly braces
- Optional return keyword

NOTE: We can pass a lambda expression wherever a functional interface is expected.

Using Lambda Expressions to create Thread

CODERINDEED

```
class MyABCD extends Thread  {
   public void run() {
          char c = 'A';
          try{      for(int i=0; i<26; i++)
                    {
                       System.out.println(c++);      Thread.sleep(500);
                    }
             }
          catch(Exception e){}
    }
}
```

Using Lambda Expressions to create Thread

CODERINDEED

```
class MyCount implements Runnable {
   public void run()
    {      try{      for(int i=1; i<11; i++)
                     {
                        System.out.println(i);
                        Thread.sleep(500);
                     }
              }
           catch(Exception e){}
    }
}
```

Using Lambda Expressions to create Thread

CODERINDEED

```
class ThreadDemo{
   public static void main(String [] rk)     {
            Thread t1 = new Thread(new MyABCD());          t1.start();
            Thread t2 = new Thread(new MyCount());         t2.start();
            Thread t3 = new Thread(
             ()->{    try{
                        for(int i=1; i<11; i++){
                              System.out.println("Threads are running");
                              Thread.sleep(1000);       }
                         }         catch(Exception e){}
               } );
            t3.start();          }
}
```

Exercise

Write the Lambda Expressions corresponding to the Functional interfaces which have following abstract methods:

A. boolean isPrime(int a)
B. double calculateSimpleInterest(int amount, double roi, int years)
C. void incrementSalary(int salary, int percentage_increment)
D. int countVowels(String s)

CODERINDEED

Programming in Java

Exception Handling

CODERINDEED

EXCEPTION

Introduction

CODERINDEED

- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- An exception is an abnormal condition that arises in a code sequence at run time.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- In other words, “An exception is a run-time error.”

Exception Handling

CODERINDEED

```
class Exception
    {
        public static void main (String arr[])
                {
                  System.out.println("Enter two numbers");
                  Scanner s = new Scanner (System.in);
                  int x = s.nextInt();
                  int y = s.nextInt();
                 int z = x/y;
                  System.out.println("result of division is : " + z);
                }
    }
```

Exception Handling

CODERINDEED

- An Exception is a run-time error that can be handled programmatically in the application and does not result in abnormal program termination.
- Exception handling is a mechanism that facilitates programmatic handling of run-time errors.
- In java, each run-time error is represented by an object.

Exception (Class Hierarchy)

CODERINDEED

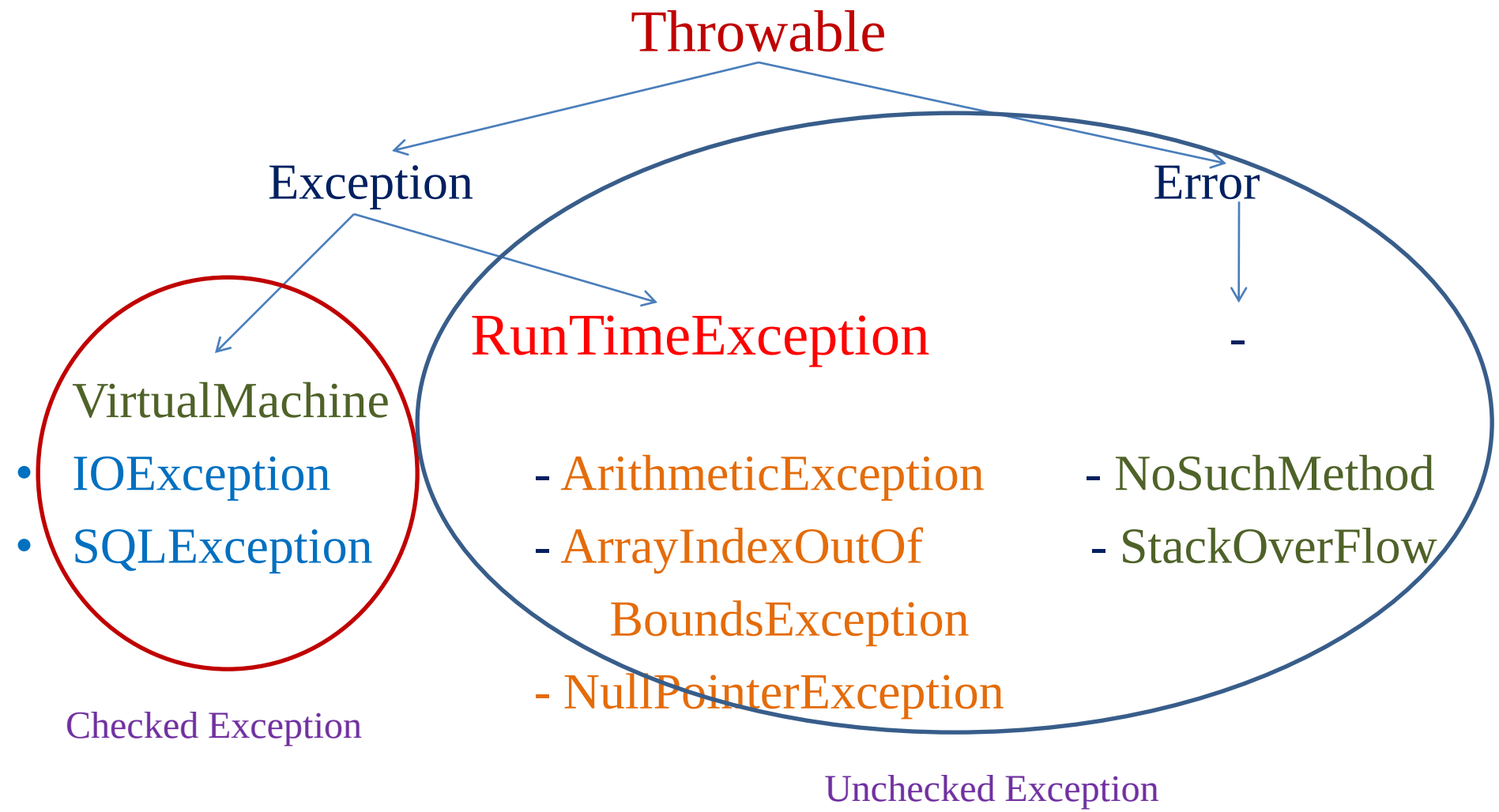
- At the root of the class hierarchy, there is a class named *'Throwable'* which represents the basic features of run-time errors.

- There are two non-abstract sub-classes of Throwable.
  - Exception : can be handled
  - Error : can't be handled

- *RuntimeException* is the sub-class of Exception.

- Each exception is a run-time error but all run-time errors are not exceptions.

CODERINDEED



Checked Exception

- Checked Exceptions are those, that have to be either caught or declared to be thrown in the method in which they are raised.

Unchecked Exception

- Unchecked Exceptions are those that are not forced by the compiler either to be caught or to be thrown.

- Unchecked Exceptions either represent common programming errors or those run-time errors that can’t be handled through exception handling.

CODERINDEED

Commonly used sub-classes of Exception

- ArithmeticException
- ArrayIndexOutOfBoundsException
- NumberFormatException
- NullPointerException
- IOException

Commonly used sub-classes of Errors

- VirualMachineError
- StackOverFlowError
- NoClassDefFoundError
- NoSuchMethodError

Uncaught Exceptions

CODERINDEED

```
class Exception_Demo
    {
        public static void main(String args[])
            {
                int d = 0;
                int a = 42 / d;
            }
    }
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception.
- once an exception has been thrown, it must be caught by an exception handler and dealt with immediately.

- In the previous example, we haven't supplied any exception handlers of our own.
- So the exception is caught by the default handler provided by the Java run-time system.
- Any exception that is not caught by your program will ultimately be processed by the default handler.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

java.lang.ArithmeticException: / by zero at Exc0.main(Exc0.java:4)

Why Exception Handling?

CODERINDEED

- When the default exception handler is provided by the Java run-time system , why Exception Handling?

- Exception Handling is needed because:
  - it allows to fix the error, customize the message .
  - it prevents the program from automatically terminating

CODERINDEED

EXCEPTION HANDLING

Keywords for Exception Handling

- try
- catch
- throw
- throws
- finally

CODERINDEED

Keywords for Exception Handling

try

- used to execute the statements whose execution may result in an exception.

```
try  {
        Statements whose execution may cause an exception
      }
```

Note: try block is always used either with catch or finally or with both.

CODERINDEED

Keywords for Exception Handling

catch

- catch is used to define a handler.
- It contains statements that are to be executed when the exception represented by the catch block is generated.
- If program executes normally, then the statements of catch block will not executed.
- If no catch block is found in program, exception is caught by JVM and program is terminated.

```
class Divide{
        public static void main(String arr[]){
            try {
                                    int a= Integer.parseInt(arr[0]);
                                    int b= Integer.parseInt(arr[1]);
                                    int c = a/b;
                                   System.out.println("Result is: "+c);
                }
            catch (ArithmeticException e)
                        {System.out.println("Second number must be non-zero");}

            catch (NumberFormatException n)
                        {System.out.println("Arguments must be Numeric");}

            catch (ArrayIndexOutOfBoundsException a)
                        {System.out.println("Invalid Number of arguments");}
            }
    }
```

Nested Try's

```
class NestedTryDemo {
    public static void main(String args[]){
      try {
            int a = Integer.parseInt(args[0]);
            try {
                  int b = Integer.parseInt(args[1]);
                  System.out.println(a/b);
                }
            catch (ArithmeticException e)
                {
                    System.out.println("Div by zero error!");
                 }
          }
    catch (ArrayIndexOutOfBoundsException) {
                    System.out.println("Need 2 parameters!");
     }
  }}
```

Defining Generalized Exception Handler

- A generalized exception handler is one that can handle the exceptions of all types.

- If a class has a generalized as well as specific exception handler, then the generalized exception handler must be the last one.

```
class Divide{
        public static void main(String arr[]){
                        try {
                                int a= Integer.parseInt(arr[0]);
                                int b= Integer.parseInt(arr[1]);
                                int c = a/b;
                               System.out.println("Result is: "+c);
                         }
                        catch (Throwable e)
                             {
                                System.out.println(e);
                             }
                }
   }
```

CODERINDEED

Keywords for Exception Handling

throw

- used for explicit exception throwing.

throw(Exception obj);

'throw' keyword can be used:

- to throw user defined exception
- to customize the message to be displayed by predefined exceptions
- to re-throw a caught exception

- Note: System-generated exceptions are automatically thrown by the Java run-time system.

```
class ThrowDemo
    {
     static void demo()
            {
             try {
                        throw new NullPointerException("demo");
                }
             catch(NullPointerException e)
                {
                        System.out.println("Caught inside demo.");
                        throw e; // rethrow the exception
                }
            }
     public static void main(String args[])
            {
             try {
                        demo();
                }
             catch(NullPointerException e)
                {
                        System.out.println("Recaught: " + e);
                }
            }
    }
```

CODERINDEED

Keywords for Exception Handling

throws

- A throws clause lists the types of exceptions that a method might throw.

```
type method-name(parameter-list) throws exception-list
        {
                // body of method
        }
```

- This is necessary for all exceptions, except those of type Error or Runtime Exception, or any of their subclasses.

- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

```
import java.io.*;
    public class ThrowsDemo
        {
            public static void main( String args []) throws IOException
                {
                    char i;
                    BufferedReader br = new BufferedReader(new
                    InputStreamReader(System.in));
                    System.out.println("Enter character, 'q' to quit");
                        do{
                            i = (char) br.read();
                            System.out.print(i);
                          }while(i!='q');
                        }
        }
```

Keywords for Exception Handling

Finally

- finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The finally block will execute whether or not an exception is thrown.
- If an exception is thrown, the finally block will execute even if no catch statement matches the exception.

- If a finally block is associated with a try, the finally block will be executed upon conclusion of the try.

- The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

CODERINDEED

Propagation of Exceptions

- If an exception is not caught and handled where it is thrown, the control is passed to the method that has invoked the method where the exception was thrown.

- The propagation continues until the exception is caught, or the control passes to the main method, which terminates the program and produces an error message.

CODERINDEED

Propagation of Exceptions

```
class ExceptionPropagation{

    public void first()     {      int data=50/0;      }

    public void second(){      first();                }

    public void third()    {   try{      second();     }
                                  catch(Exception e){
                                          System.out.println(“Done");}
                           }

    public static void main(String [] rk){
    ExceptionPropagation ob = new ExceptionPropagation();
            ob.third();
            System.out.println(“Thank You");
            }
 }
```

CODERINDEED

Defining Custom Exceptions

- We can create our own Exception sub-classes by inheriting Exception class.
- The Exception class does not define any methods of its own.
- It inherits those methods provided by Throwable.
- Thus, all exceptions, including those that we create, have the methods defined by Throwable available to them.

Constructor for creating Exception

- Exception( )
- Exception(String msg)

- A custom exception class is represented by a subclass of *Exception / Throwable.*

- It contains the above mentioned constructor to initialize custom exception object.

```
class Myexception extends Throwable
    {
     public Myexception(int i)
       {
       System.out.println("you  have entered ." +i +"  : It exceeding
    the limit");
       }
     }
```

CODERINDEED

```
public class ExceptionTest {
    public void show(int i) throws Myexception {
                        if(i>100)
                throw new Myexception(i);
                        else
                System.out.println(+i+" is less then 100 it is ok");
        }
     public static void main(String []args){
                int i=Integer.parseInt(args[0]);
                int j=Integer.parseInt(args[1]);
                ExceptionTest t=new ExceptionTest();
                try{
                                        t.show(i);  t.show(j);
                        }
                catch(Throwable e) {
                                System.out.println("catched exception is "+e);
                        }
        }
}
```

CODERINDEED

Try with Resource Statement

- JDK 7 introduced a new version of try statement known as try-with-resources statement.
- This feature add another way to exception handling with resources management,
- It is also referred to as automatic resource management.

Syntax:

```
try(resource-specification)
        { //use the resource }
catch(Exception ex) {...}
```

CODERINDEED

Try with Resource Statement

- The try statement contains a parenthesis in which one or more resources is declare.

- Any object that implements java.lang.AutoCloseable or java.io.Closeable, can be passed as a parameter to try statement.
- A resource is an object that is used in program and must be closed after the program is finished.

- The try-with-resources statement ensures that each resource is closed at the end of the statement, you do not have to explicitly close the resources.

Multi-catch

```
try{
        x = Integer.parseInt(rk[0]);
        y = Integer.parseInt(rk[1]);
    }
  catch(ArrayIndexOutOfBoundsException | NumberFormatException ae)
    {
        if( ae instanceof NumberFormatException)
          {
                System.out.println("Enter the Integers");
                ae = new ArithmeticException();
          }
        else if( ae instanceof ArrayIndexOutOfBoundsException)
                System.out.println("Enter Two Integers");

    }
```

CODERINDEED

Programming in Java

Files and I/O Streams

By

coderindeed.in

Introduction

CODERINDEED

- Most real applications of Java are not text-based, console programs.
- Java's support for console I/O is limited.
- Text-based console I/O is not very important to Java programming.
- Java does provide strong, flexible support for I/O as it relates to files and networks.

CODERINDEED

FILES

File Class

CODERINDEED

- The File class provides the methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory.

- An absolute file name (or full name) contains a file name with its complete path and drive letter.
- For example, c:\book\Welcome.java

- A relative file name is in relation to the current working directory.
- The complete directory path for a relative file name is omitted.
- For example, Welcome.java

File Class

CODERINDEED

- The File class is a wrapper class for the file name and its directory path.

- For example, new File("c:\\book")

  creates a File object for the directory c:\book,

- new File("c:\\book\\test.dat")

  creates a File object for the file c:\book\test.dat

- File class does not contain the methods for reading and writing file contents.

Methods and Constructors

CODERINDEED

Constructor:

File(String path_name)

- Creates a File object for the specified path name. The path name may be a directory or a file.

Methods of File Class

CODERINDEED

Methods:

boolean isFile()
boolean isDirectory()
boolean isHidden()
boolean exists()
boolean canRead()
boolean canWrite()
String getName()
String getPath()
String getAbsolutePath()
long lastModified()
long length()
boolean delete()
boolean renameTo(File f)
File [] listFiles()

Important

- The lastModified() method returns the date and time when the file was last modified.
- It is measured in milliseconds since the beginning of Unix time (00:00:00 GMT, January 1, 1970).
- The Date class is used to display it in a readable format.

```
System.out.println("Last modified on " +
                        new java.util.Date(file.lastModified()));
```

Reading and Writing Files

CODERINDEED

- In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file.
- Java allows us to wrap a byte-oriented file stream within a character-based object.
- We can use Scanner and PrintWriter class to read and write Files.

PrinterWriter Class

CODERINDEED

- Although using System.out to write to the console is acceptable.
- The recommended method of writing to the console when using Java is through a PrintWriter stream.
- PrintWriter is one of the character-based classes.
- Using a character-based class for console output makes it easier to internationalize your program.

PrintWriter(OutputStream outputStream, boolean flushOnNewline)

- outputStream is an object of type OutputStream.
- flushOnNewline controls whether Java flushes the output stream every time a println( )method is called.
- If flushOnNewline is true, flushing automatically takes place.
- If false, flushing is not automatic.
- PrintWriter supports the print( ) and println( ) methods for all types including Object. Thus, we can use these methods in the same way as they have been used with System.out.
- If an argument is not a simple type, the PrintWriter methods call the object's toString( ) method and then print the result.

Using PrintWriter

```
import java.io.*;
public class PrintWriterDemo
    {
        public static void main(String args[])
            {
                PrintWriter pw = new PrintWriter(System.out, true);
                pw.println("Using PrintWriter Object");
                int i = -7;
                pw.println(i);
                double d = 4.5e-7;
                pw.println(d);
            }
    }
```

PrintWriter Methods

CODERINDEED

java.io.PrintWriter

```
+PrintWriter(file: File)
+PrintWriter(filename: String)
+print(s: String): void
+print(c: char): void
+print(cArray: char[]): void
+print(i: int): void
+print(l: long): void
+print(f: float): void
+print(d: double): void
+print(b: boolean): void
```

Also contains the overloaded `println` methods.

Also contains the overloaded `printf` methods.

Using Scanner

- The java.util.Scanner class is used to read strings and primitive values from the console.
- Scanner breaks the input into tokens delimited by whitespace characters.
- To read from the keyboard, we create a Scanner as follows:

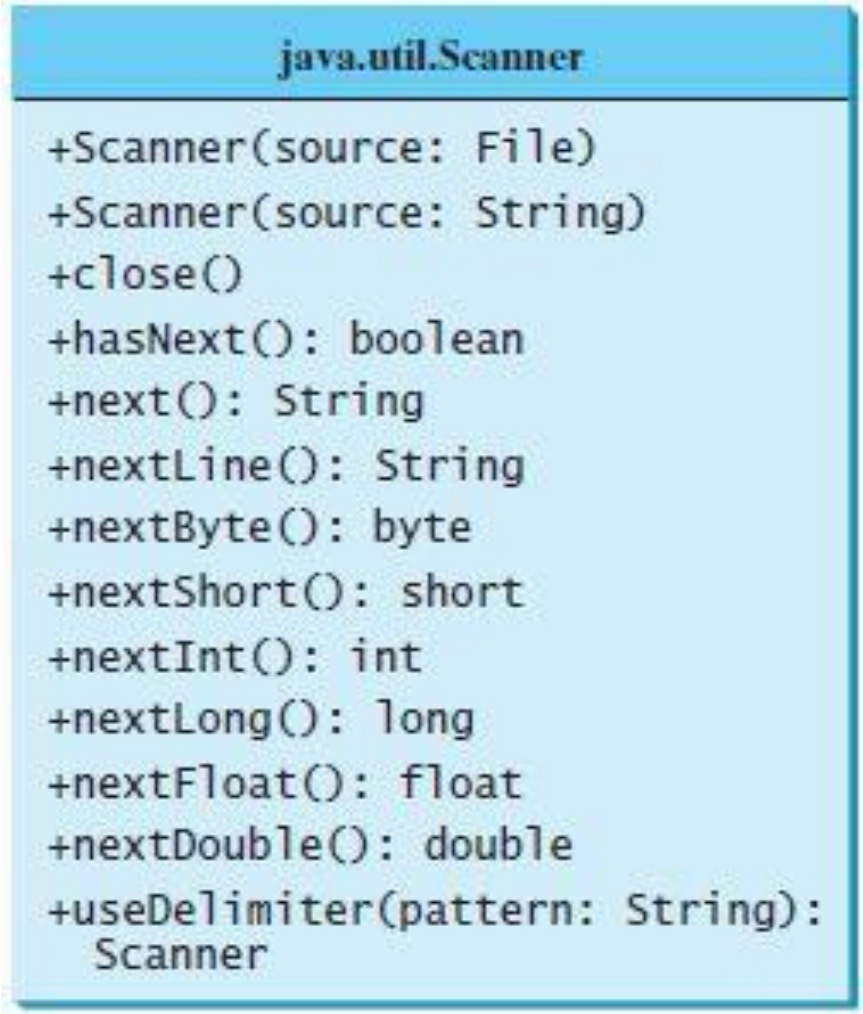
```
Scanner s = new Scanner(System.in);
```

- To read from a file, create a Scanner for a file, as follows:

```
Scanner s = new Scanner(new File(filename));
```

Scanner Methods

CODERINDEED



CODERINDEED

I/O STREAMS

Streams

- Java implements streams within class hierarchies defined in the java.io package.
- *A stream is an ordered sequence of data.*
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.
- An I/O Stream represents an input source or an output destination.

I/O Streams

CODERINDEED

- A stream can represent many different kinds of sources and destinations

  – disk files, devices, other programs, a network socket, and memory arrays

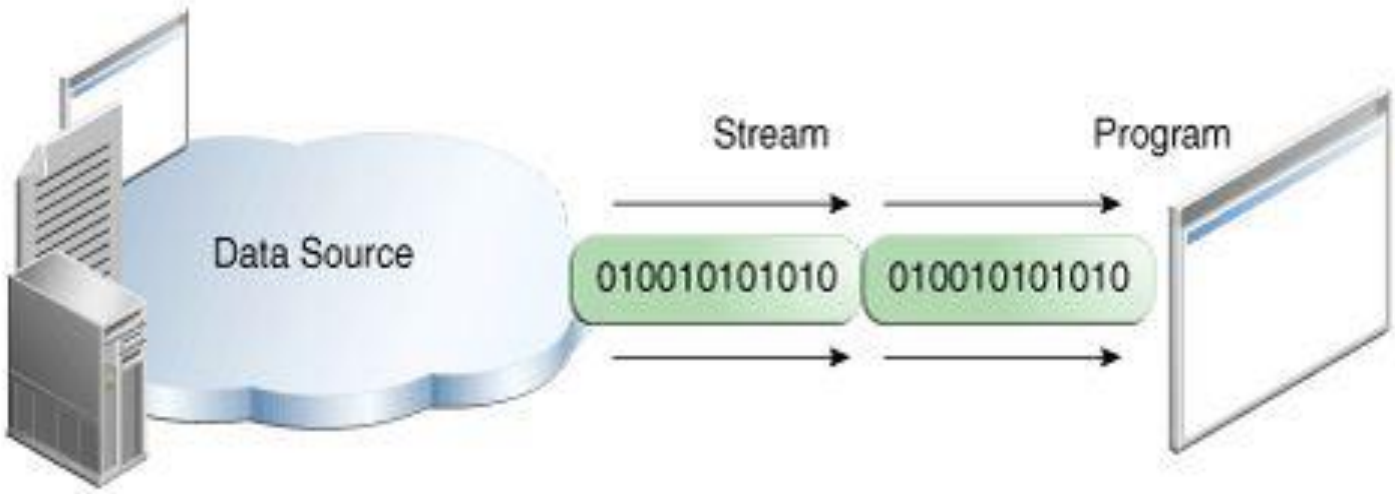
- Streams support many different kinds of data

  – simple bytes, primitive data types, localized characters, and objects

- Some streams simply pass on data; others manipulate and transform the data in useful ways.

Input Stream

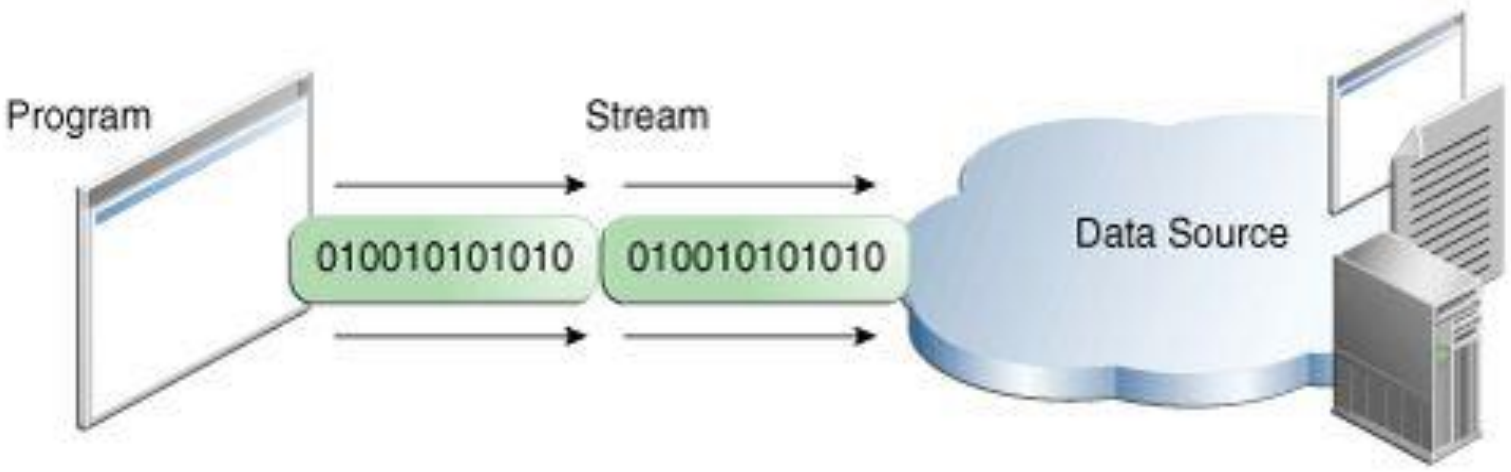
- A program uses an input stream to read data from a source, one item at a time.



Reading information into a program.

Output Stream

- A program uses an *output stream* to write data to a destination, one item at time:



Writing information from a program.

Types of Streams

CODERINDEED

- Java defines two different types of Streams-
  - *Byte Streams*
  - *Character Streams*
- Byte streams provide a convenient means for handling input and output of bytes.
- Byte streams are used, for example, when reading or writing binary data.
- Character streams provide a convenient means for handling input and output of characters.
- In some cases, *character streams are more efficient than byte streams*.

Byte Streams

CODERINDEED

- Programs use *byte streams* to perform input and output of 8-bit bytes.
- Byte streams are defined by using two class hierarchies.
- At the top, there are two abstract classes: InputStream and OutputStream.
- The abstract classes InputStream and OutputStream define several key methods that the other stream classes implement.
- Two of the most important methods are read( )and write( ), which, respectively, read and write bytes of data.
- Both methods are declared as abstract inside InputStream and OutputStream.

Closing the Streams

CODERINDEED

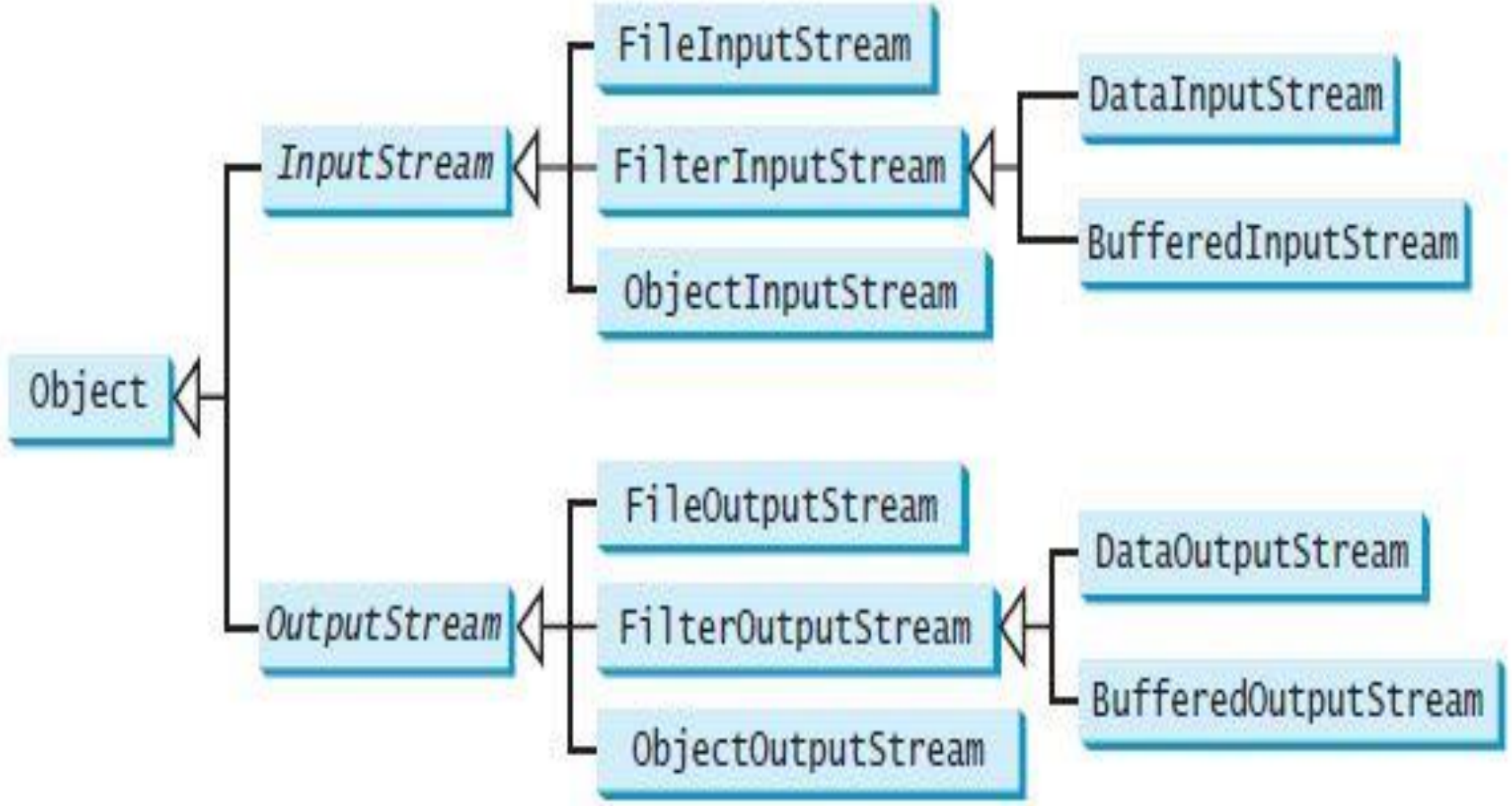
- Closing a stream when it's no longer needed is very important.
- It is so important that we have used a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.
- When we call the close(), it clears the buffer by performing the write operation to the destination and then closes the stream.
- That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

CODERINDEED

BINARY I/O CLASSES

CODERINDEED

Binary Input/Output Classes



Byte Stream Classes

CODERINDEED

Stream Class	Meaning / Use
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
DataInputStream	contains methods for reading the Java standard data types
DataOutputStream	contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PrintStream	Output stream that contains print() and println( )
PipedInputStream	Input Pipe
PipedOutputStream	Output Pipe

©CODERINDEED

Methods defined by 'InputStream'

Method	Description
int available( )	Returns the number of bytes of input currently available for reading.
void close( )	Closes the input source. Further read attempts will generate an **IOException**.
void mark(int *numBytes*)	Places a mark at the current point in the input stream that will remain valid until *numBytes* bytes are read.
boolean markSupported( )	Returns **true** if **mark( )/reset( )** are supported by the invoking stream.
int read( )	Returns an integer representation of the next available byte of input. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ])	Attempts to read up to *buffer.length* bytes into *buffer* and returns the actual number of bytes that were successfully read. –1 is returned when the end of the file is encountered.
int read(byte *buffer*[ ], int *offset*, int *numBytes*)	Attempts to read up to *numBytes* bytes into *buffer* starting at *buffer*[*offset*], returning the number of bytes successfully read. –1 is returned when the end of the file is encountered.
void reset( )	Resets the input pointer to the previously set mark.
long skip(long *numBytes*)	Ignores (that is, skips) *numBytes* bytes of input, returning the number of bytes actually ignored.

Methods defined by 'OutputStream'

CODERINDEED

Method	Description
void close( )	Closes the output stream. Further write attempts will generate an **IOException**.
void flush( )	Finalizes the output state so that any buffers are cleared. That is, it flushes the output buffers.
void write(int *b*)	Writes a single byte to an output stream. Note that the parameter is an **int**, which allows you to call **write( )** with expressions without having to cast them back to **byte**.
void write(byte *buffer*[ ])	Writes a complete array of bytes to an output stream.
void write(byte *buffer*[ ], int *offset*, int *numBytes*)	Writes a subrange of *numBytes* bytes from the array *buffer*, beginning at *buffer*[*offset*].

FileInputStream and FileOutputStream

CODERINDEED

- *FileInputStream* and *FileOutputStream* are stream classes which create byte streams linked to files.

FileInputStream(String fileName) throws FileNotFoundException

FileInputStream(File f) throws FileNotFoundException

FileOutputStream(String fileName) throws FileNotFoundException

FileOutputStream(File f) throws FileNotFoundException

FileOutputStream(String fileName, boolean append) throws FileNotFoundException

FileOutputStream(File f, boolean append) throws FileNotFoundException

Reading and Writing Files

CODERINDEED

- To read from a file, we can use read( ) that is defined with in FileInputStream.

int read( ) throws IOException

- Each time read() is called, it reads a single byte from the file and returns the byte as an integer value.

- To write to a file, we can use the write( )method defined by FileOutputStream.

void write(int byteval) throws IOException

```
import java.io.*;
class CopyFile {
    public static void main(String args[])throws IOException{
            int i; FileInputStream fin=null; FileOutputStream fout=null;
            fin = new FileInputStream(args[0]);
            fout = new FileOutputStream(args[1]);
       try {
             do {       i = fin.read();
                        if(i != -1) fout.write(i);
                  }     while(i != -1);
            }
      catch(IOException e)    {
                                 System.out.println("File Error");
                                }
      fin.close();
      fout.close();
   }
  }
```

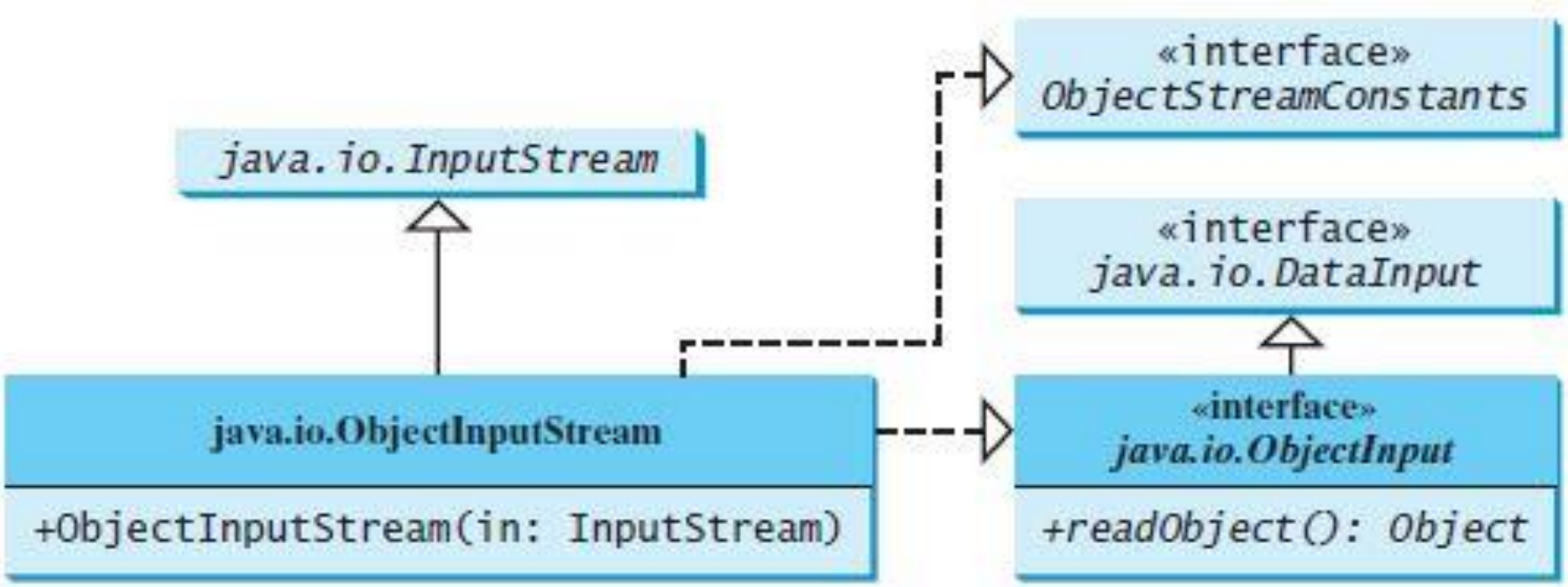
Object Input/Output

CODERINDEED

- ObjectInputStream/ObjectOutputStream classes can be used to read/write *serializable objects*.
- ObjectInputStream/ObjectOutputStream enables you to perform I/O for objects in addition to primitive type values and strings.
- ObjectInputStream/ObjectOutputStream contains all the functions of DataInputStream/ DataOutputStream.

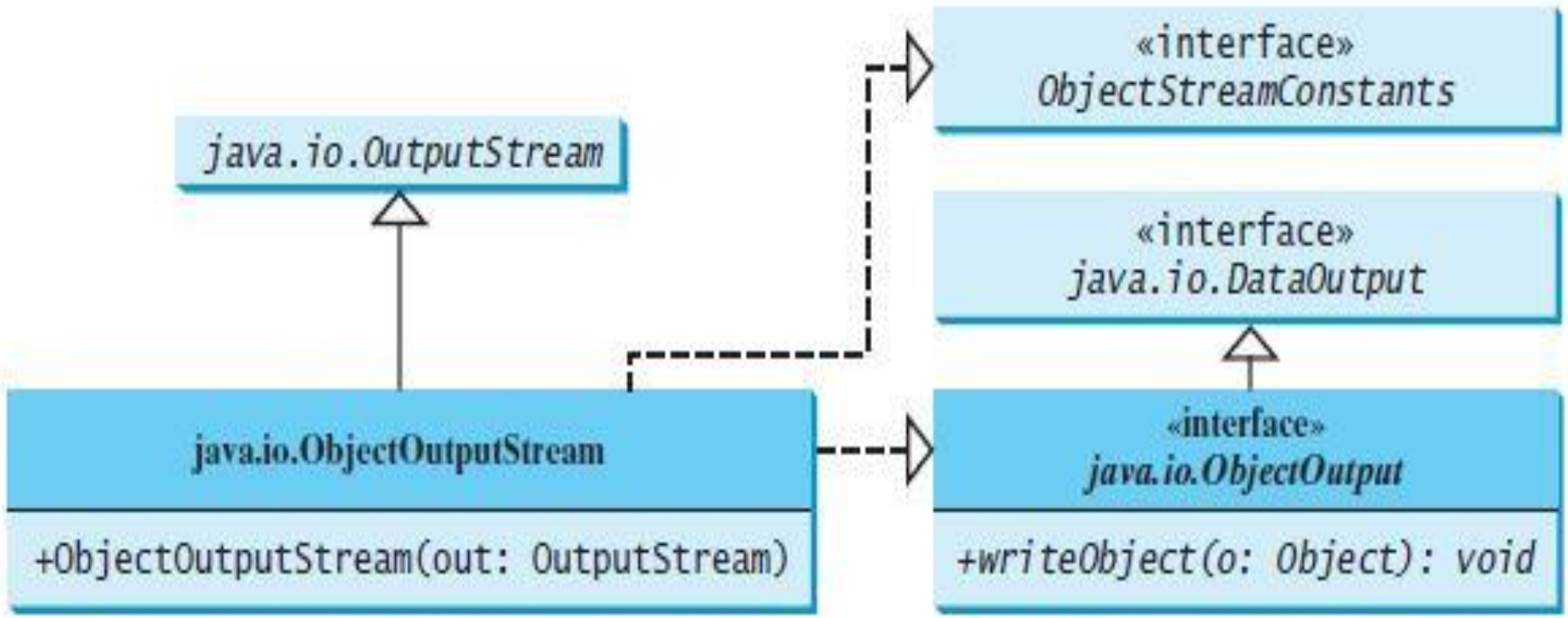
CODERINDEED

Constructor and Methods of ObjectInputStream



CODERINDEED

Constructor and Methods of ObjectOutputStream



SERIALIZATION

Serialization

CODERINDEED

- Serialization is the process of writing the state of an object to a byte stream.
- This is useful when we want to save the state of our program to a persistent storage area, such as a file.
- At a later time, we may restore these objects by using the process of de-serialization.
- Serialization is also needed to implement Remote Method Invocation (RMI).

- An object to be serialized may have references to other objects, which, in turn, have references to still more objects.

- If we attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized.

Serializable Interface

CODERINDEED

- Only an object that implements the Serializable interface can be saved and restored by the serialization facilities.
- The Serializable interface defines no members.
- It is simply used to indicate that a class may be serialized.
- If a class is serializable, all of its subclasses are also serializable.

Serialization Example

CODERINDEED

```
class MyClass implements Serializable
    {
        String s;
        int i;
        double d;
    public MyClass(String s, int i, double d) {
        this.s = s;
        this.i = i;
        this.d = d;
      }
    public String toString()
        {
            return "s=" + s + "; i=" + i + "; d=" + d;
        }
    }
```

```
import java.io.*;
public class SerializationDemo {
    public static void main(String args[]) {
        try {
            MyClass object1 = new MyClass("Hello", -7, 2.7e10);
            System.out.println("object1: " + object1);
            FileOutputStream fos = new FileOutputStream("serial.txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(object1);
            oos.flush();
            oos.close();
        }
        catch(IOException e) {
            System.out.println("Exception during serialization: " + e);
            System.exit(0);
        }
```

```
// Object deserialization

try {
    MyClass object2;
    FileInputStream fis = new FileInputStream("serial.txt");
    ObjectInputStream ois = new ObjectInputStream(fis);
    object2 = (MyClass)ois.readObject();
    ois.close();
    System.out.println("object2: " + object2);
  }
catch(Exception e) {
    System.out.println("Exception during deserialization: " + e);
    System.exit(0);
   }
  }
}
```

Exercise

Write a program which prompts the user to enter the path of the file to be read (f1) and file to be written(f2).

Append the content of the file f1 at the end of file f2.

Write a program to define a class Car having attributes name, mileage and price. Create 5 Car objects and write those objects in a file named as mycar.doc.

Write a method public void getCars(double mlg) which displays the name, price and mileage of all the cars having mileage greater than mlg.

CODERINDEED

Programming in Java

Topic: Assertion

By

coderindeed.in

Contents

Introduction

- An assertion is a statement in Java that enables you to test your assumptions about the program.
- Each assertion contains a boolean expression that you believe will be true when the assertion executes.
- If it is not true, the system will throw a runtime error (java.lang.AssertionError).

Example

For example, if we write a method that calculates the interest of any account, we might assert that the roi or amount is positive.

```
double calculateInterest(int amt, double roi, int years)
{
        //Using assertion
        assert amt>0;
        double interest = (amt*roi*years)/100;
        return interest;
}
```

Assertion

- The assertion statement has two forms.
  - assert *Expression1*;

    Here, Expression1 is a boolean expression. When the system runs the assertion, it evaluates Expression1 and if it is false throws an AssertionError with no detail message.

  - assert *Expression1* : *Expression2*;

    Here, Expression1 is a boolean expression and Expression2 is an expression that has a value and it cannot be an invocation of a method that is declared void.

Example

```
public class AssertionExample {
    public static void main(String[] rk) {
        // get a number in the first argument
        int num = Integer.parseInt(rk[0]);

          assert num <= 10;          // stops if number > 10
          assert num >20 : “Number is greater than 20”;

         System.out.println("Pass");
    }
}
```

Using Assertion

Assertion will not work directly because assertion is disabled by default.

To enable the assertion, -ea or -enableassertions switch of java must be used.

Compile: javac AssertionExample.java

Run: java -ea AssertionExample

OR

java -enableasserions AssertionExample

How Assertion Works?

In assertion a BooleanExpression is used and if boolean expression is false then java throws AssertionError and the program terminates.

Second form of assert keyword "assert booleanExpression : errorMessage" is more useful and provides a mechanism to pass additional data when Java assertion fails and java throws AssertionError.

Need of Assertion

```
class AssertionNeed{
   public static void main(String [] rk)
{                    new AssertionNeed().remainder(8);    }
   void remainder(int i)
   {        if(i%3==0) System.out.println("Divisible by 3");
            else if(i%3==1) System.out.println("Remainder 1");
            else{
                System.out.println("Remainder 2");   }
   }
}
```

What About it?

```
class AssertionNeedTest
   {
         public static void main(String [] rk)
               {
                     new AssertionNeed().remainder(-7);
               }
   }
```

Need of Assertion

```
class AssertionNeed{
   public static void main(String [] rk)
{                    new AssertionNeed().remainder(-7);    }
   void remainder(int i)
   {        if(i%3==0) System.out.println("Divisible by 3");
            else if(i%3==1) System.out.println("Remainder 1");
            else{
                  // assert i%3 == 2 : i;
                    System.out.println("Remainder 2");   }
   }
}
```

Why Assertion?

- Using assertion helps to detect bug early in development cycle which is very cost effective.
- Assertion in Java makes debugging easier as AssertionError is a best kind of error while debugging because its clearly tell source and reason of error.
- Assertion is great for data validation if by any chance your function is getting incorrect data your program will fail with AssertionError.

Important

- Assertion is introduced in JDK 1.4 and implemented using assert keyword in java.
- Assertion can be enabled at run time by using switch –ea or –enableassertion
- Assertion does not replace Exception but compliments it.
- Do not use assertion to validate arguments or parameters of public method.

Let's Do Something

Write a program which prompts the user to enter the amount, rate of interest and years and calculate the interest using the following method:

public static double calculateInterest(int amt, double roi, int years)

Use assertions to check the basic requirements of the parameters in the method and display the updated balance.

CODERINDEED

Programming in Java

Topic: Generics

By

coderindeed.in

Contents

Introduction

- Enables to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter.
- Introduced in Java by jdk 1.5.
- Generics means parameterized types.
- Generics add the type-safety.
- Generics add stability to your code by making more of your bugs detectable at compile time.

Why Generics?

- The functionality of Gen class can be achieved without generics by specifying Object type and using proper casting whenever required.

Then why we use Generics?

- Java compiler does not have knowledge about the type of data actually stored in NonGen. So-
- Explicit casts must be employed to retrieve the stored data.
- Several type mismatch errors cannot be found until run time.

Why Generics?

- Stronger type checks at compile time
- Elimination of casts

```
List list = new ArrayList(); list.add("hello");
String s = (String) list.get(0);
```

- **Using generics:**

```
List<String> list = new ArrayList<String>(); list.add("hello");
String s = list.get(0); // no cast
```

- Enabling programmers to implement generic algorithms.
  We can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Advantage of Generics

- The ability to create type-safe code in which type-mismatch errors are caught at compile time is a key advantage of generics.

Generics Work Only with Objects

- When declaring an instance of a generic type, the type argument passed to the type parameter must be a class type.

```
Gen<int> strOb = new Gen<int>(53);
```

- The above declaration is an error.

- A reference of one specific version of a generic type is not type compatible with another version of the same generic type.

```
iOb = strOb;                    // Wrong!
```

GENERIC CLASS

General Form of Generic Class

- The generics syntax for declaring a generic class:

  class class-name<type-param-list>

  { // ... }

- The syntax for declaring a reference to a generic class:

  class-name<type-arg-list> var-name =

  new *class-name<type-arg-list>(cons-arg-list);*

Generic Class with Multiple Type Parameters

```
class TwoGen<T, V> {
        T ob1;  V ob2;
        TwoGen(T o1, V o2) {
                ob1 = o1; ob2 = o2; }
        void showTypes() {
                System.out.println("Type of T is " +
                                ob1.getClass().getName());
                System.out.println("Type of V is " +
                                ob2.getClass().getName()); }
        T getob1() { return ob1; }
        V getob2() { return ob2; }
}
```

```
class SimpGen {
        public static void main(String args[]) {
                TwoGen<Integer, String> t =
                new TwoGen<Integer, String>(16920,  "Ravi
  Kant");
                t.showTypes();
                int v = t.getob1();
                System.out.println("value: " + v);
                String str = t.getob2();
                System.out.println("value: " + str);
        }
  }
```

Bounded Types

- Used to limit the types that can be passed to a type parameter.

- When specifying a type parameter, we can create an upper bound that declares the super-class from which all type arguments must be derived.

<T extends superclass>

- A bound can include both a class type and one or more interfaces.

class Gen<T extends MyClass **&** MyInterface>

Generic Interfaces

- Generic interfaces are specified just like generic classes.

Example:

```
interface MinMax<T extends Comparable<T>>
        {  T min();     T max(); }
```

- The implementing class must specify the same bound.

- Once the bound has been established, it need not to be specified again in the implements clause.

Generic Interface

```
interface MinMax<T extends Comparable<T>>
    {
    T min();
    T max();
    }

class My<T extends Comparable<T>> implements MinMax<T>
    {
            …
    }
```

CODERINDEED

Programming in Java

Topic: Collection Framework

By

coderindeed.in

Contents

Introduction

- A collection is simply an object that groups multiple elements into a single unit.
- Collections are used to store, retrieve, manipulate, and communicate aggregate data.
- *java.util* package contains all the collection classes and interfaces.

Collection Framework

- A collections framework is a unified architecture for representing and manipulating collections.
- It is a collection of classes and interfaces.
- At the top of collection framework hierarchy, there is an interface, *Collection*.

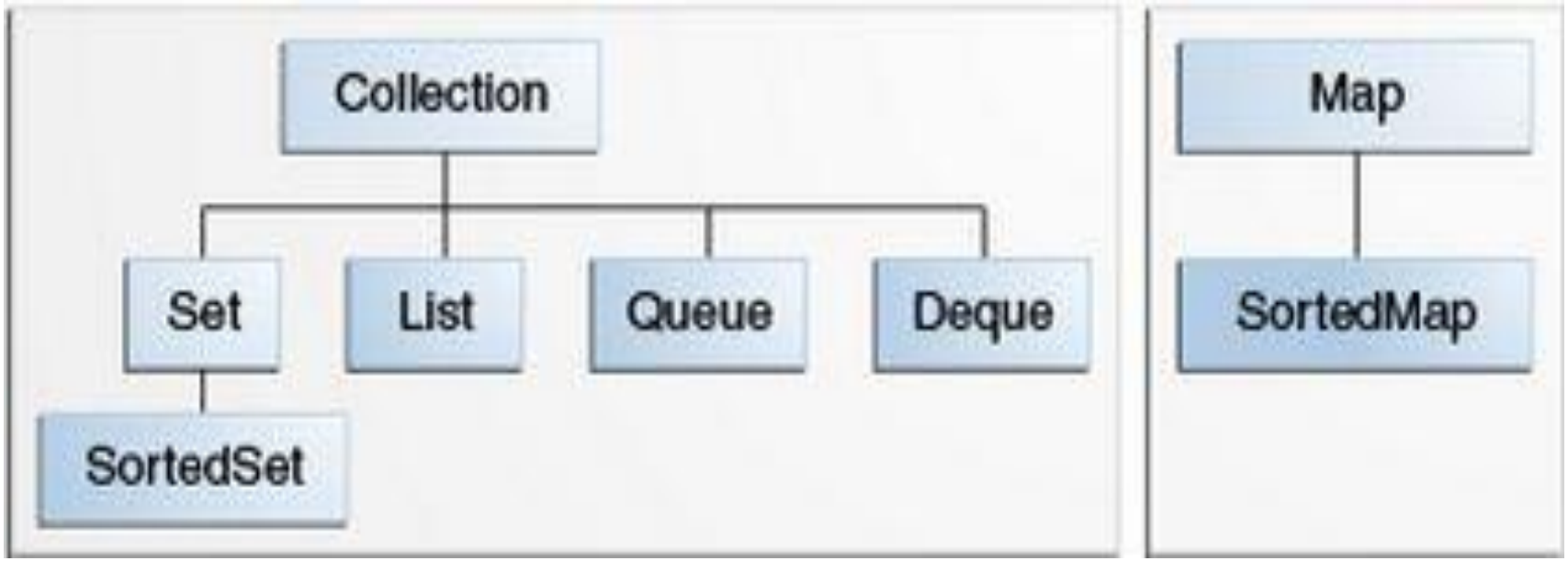
Advantages of Collections

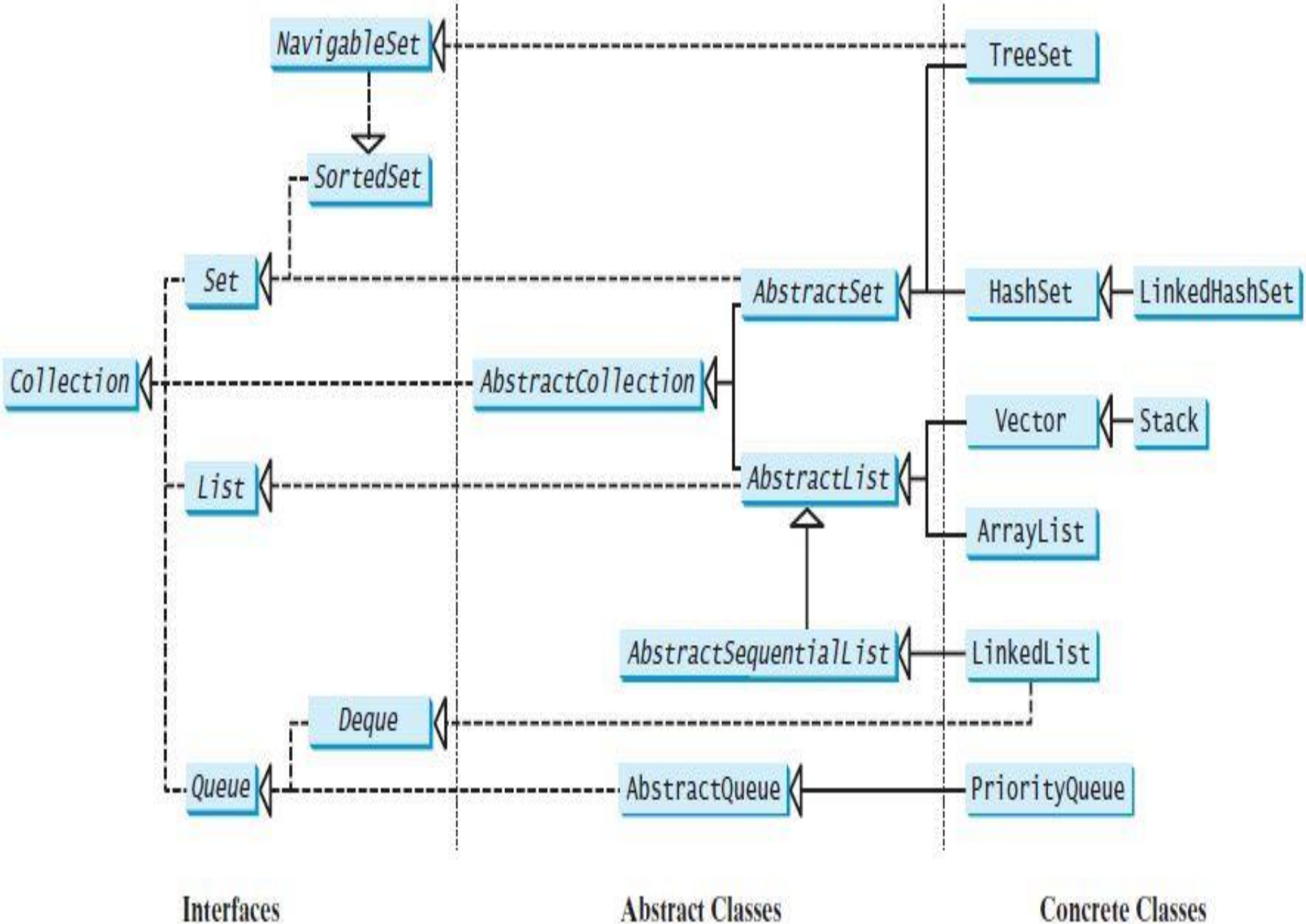
- Reduces programming effort
- Increases program speed and quality
- Allows interoperability among unrelated APIs

Collection Interfaces

The Java Collections Framework supports two types of containers:

1. Collection: for storing a collection of elements
2. Map: for storing key/value pairs




NavigableSet
TreeSet
SortedSet
Set
AbstractSet
HashSet
LinkedHashSet
Collection
AbstractCollection
Vector
Stack
List
AbstractList
ArrayList
AbstractSequentialList
LinkedList
Deque
Queue
AbstractQueue
PriorityQueue
Interfaces
Abstract Classes
Concrete Classes

Collection Interfaces

- Sets store a group of non-duplicate elements.
- Lists store an ordered collection of elements.
- Queues store objects that are processed in first-in, first-out fashion.
- Maps are efficient data structures for quickly searching an element using a key.

- The *AbstractCollection* class provides partial implementation for the Collection interface.

- It implements all the methods in Collection except the *size()* and *iterator()* methods.

COLLECTION

INTERFACE

Methods of Collection Interface

boolean add(Object o)

- Appends the specified element o to the end of the collection.

boolean addAll(Collection c)

- Appends all of the elements in the specified collection to the end of the collection.

boolean contains(Object o)

- Returns true if this list contains the specified element.

boolean containsAll(Collection c)

- Returns true if this collection contains all the elements in c.

boolean remove(Object o)

- Removes the object from this collection.

boolean removeAll(Collection c)

- Removes all the elements in c from this collection.

boolean retainAll(Collection c)

- Retains the elements that are both in c and in this collection.

void clear()

- Removes all of the elements from the collection.

boolean isEmpty()

- Returns true if this collection contains no elements.

int size()

- Returns the number of elements in the collection.

Object [] toArray()

- Returns an array of Object for the elements in the collection.

int hashCode()

- Returns the hash code for the collection.

Iterator iterator()

- Returns the iterator object for the invoking collection.

- I

Practice MCQs

Which interface restricts duplicate elements?

A. Set
B. List
C. Map
D. All of these

Which does NOT inherit the Collection interface?

A. List
B. Map
C. Set
D. None of these

Practice MCQs

Deque and Queue are derived from:

A. AbstractList

B. Collection

C. AbstractCollection

D. List

The root interface of Java Collection framework hierarchy is -

A. Collection

B. List

C. Collections

D. Set

Practice MCQs

LinkedList class implements?

A.Deque

B.List

C.Both A and B

D.None of these

ITERATORS

Iterator

- There is an Iterable interface at the top of Collection hierarchy.
- The Collection interface extends the java.lang.Iterable interface.
- It defines a method which returns an Iterator object for the elements of Collection.

  Iterator iterator()

- *Iterator* object is used to traverse the elements in a collection.

Methods of Iterator Interface

boolean hasNext()

Returns true if this iterator has more elements to traverse.

Object next()

Returns the next element from this iterator.

void remove()

Removes the last element obtained using the next method.

ListIterator

ListIterator

- ListIterator is derived from Iterator interface and comes with more methods than Iterator.
- ListIterator can be used to traverse for List type Objects.
- Allows to traverse in both the directions.

ListIterator

- A ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next().

- The remove() and set(Object) methods are not defined in terms of the cursor position; they are defined to operate on the last element returned by a call to next() or previous().

ListIterator Methods

- public abstract boolean hasPrevious()
- public abstract Object previous();

- public abstract int nextIndex();
- public abstract int previousIndex();

- public abstract void set(Object);
- public abstract void add(Object);

LIST INTERFACE

List

- The List interface *extends the Collection* interface.
- It defines a collection for storing elements in a sequential order.
- It define an *ordered collection* with *duplicates allowed*.
- *ArrayList* , *Vector* and *LinkedList* are the sub-classes of List interface.

Methods of List Interface

- boolean add(int index, Object o)
- boolean addAll(int index, Collection c)
- Object remove(int index)
- Object get(int index)
- Object set(int index, Object o)
- int indexOf(Object o)
- int lastIndexOf(Object o)
- ListIterator listIterator()
- ListIterator listIterator(int start_index)
- List subList(int start_index, int end_index): returns a sublist from start_index to end_index-1

Sub-classes of List

- ArrayList
- LinkedList
- Vector
- Stack

ARRAYLIST

ArrayList

- The ArrayList class extends *AbstractList* and implements the *List* interface.
- Defined in *java.util* package.
- ArrayList supports dynamic arrays that can grow as needed.
- Array lists are created with an initial size.
- When this size is exceeded, the collection is automatically enlarged.
- When objects are removed, the array may be shrunk.

ArrayList Constructors

- T he ArrayList class supports three constructors.

- ArrayList() : creates an empty array list with an initial capacity sufficient to hold 10 elements.

- ArrayList(int capacity) : creates an array list that has the specified initial capacity.

- ArrayList(Collection c) : creates an array list that is initialized with the elements of the collection c.

Methods of ArrayList

boolean add(Object o)

- Appends the specified element to the end of this list.

void add(int index, Object element)

- Inserts the specified element at the specified position index in this list.
- Throws IndexOutOfBoundsException if the specified index is out of range.

boolean addAll(Collection c)

- Appends all of the elements in the specified collection to the end of this list.
- Throws NullPointerException if the specified collection is null.

boolean addAll(int index, Collection c)

- Inserts all of the elements of the specified collection into this list, starting at the specified position.
- Throws NullPointerException if the specified collection is null.

Object remove(int index)

- Removes the element at the specified position in this list.

Object remove(Object o)

- Removes the element o from this list.

Object clone()

- Returns a shallow copy of this ArrayList.

int size()

- Returns the number of elements in the list.

void clear()

- Removes all of the elements from this list.

boolean contains(Object o)

- Returns true if this list contains the specified element.

Object get(int index)

- Returns the element at the specified position in this list.

int indexOf(Object o)

- Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain the element.

int lastIndexOf(Object o)

- Returns the index in this list of the last occurrence of the specified element, or -1.

void ensureCapacity(int minCapacity)

- Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

Object set(int index, Object element)

- Replaces the element at the specified position in this list with the specified element.
- Throws IndexOutOfBoundsException if the specified index is out of range (index < 0 || index >= size()).

void trimToSize()

- Trims the capacity of this ArrayList instance to be the list's current size.

SET INTERFACE

Set Interface

- A set is an efficient data structure for storing and processing *non-duplicate elements*.
- Set does not introduce new methods or constants.
- The sub-classes of Set (HashSet, TreeSet & LinkedHashSet) ensure that no duplicate elements can be added to the set.

Set Interface

- The *AbstractSet* class provides implementations for the *equals()* method and the *hashCode()*.
- The hash code of a set is the sum of the hash codes of all the elements in the set.
- The *size()* and *iterator()* are not implemented in the AbstractSet class.

TREESET

TreeSet

- TreeSet implements the NavigableSet interface.
- We can add objects into a tree set as long as they can be compared with each other.
- Access and retrieval times are quite fast.

TreeSet

- Constructor:

  TreeSet()

  TreeSet(Collection c)

  TreeSet(Comparator c)

Comparator Interface

- java.util.Comparator interface provides two abstract methods:

```
public interface Comparator<T>
        {
          public abstract int compare(T ob1, T ob2);
          public abstract boolean equals(java.lang.Object);
          …
        }
```

Comparable Interface

- java.lang.Comparable interface provides only one abstract method:

```
public interface Comparable<T>
        {
          public abstract int compareTo(T ob);
        }
```

Implementing TreeSet

```
import java.util.*;
public class SortSetExample {
   public static void main(String...rk){
       Set<Integer> tree = new TreeSet<Integer>();
       tree.add(3);
       tree.add(1);
       tree.add(2);
       System.out.println("treeSet : "+treeSet);

   }
}
```

Implementing TreeSet with Comparator sorting in descending order

CODERINDEED

```
import java.util.*;
public class SortSetExample {
   public static void main(String...rk){
        Set<Integer> tree = new TreeSet<Integer>(new
                    Comparator<Integer>() {
                        public int compare(Integer o1, Integer o2) {
                            return o2.compareTo(o1);
                        }
                    });
       tree.add(3);          tree.add(1);                tree.add(2);
       System.out.println("treeSet : "+tree);
   }
}
```

Implementing TreeSet with Custom Objects using Comparator

CODERINDEED

```
class Employee{
   String name;
   String id;
   public Employee(String name, String id) {
      this.name = name;       this.id = id;
   }

   public String toString() {
      return "Employee{" + "name=" + name + ", id=" + id  + '}';
   }
}
```

Implementing TreeSet with Custom Objects using Comparator

CODERINDEED

```
public class SortSetExample {
   public static void main(String...a){
      Employee emp1=new Employee("sam","4");
      Employee emp2=new Employee("amy","2");
      Employee emp3=new Employee("brad","1");
     Set<Employee> treeSet = new TreeSet<Employee>(new
    Comparator<Employee>() {
                    public int compare(Employee o1, Employee o2)
                          { return o1.name.compareTo(o2.name); }
                   });
treeSet.add(emp1);      treeSet.add(emp2);      treeSet.add(emp3);
      System.out.println("treeSet : "+treeSet);
   }}
```

DEQUEUE

Deque Interface

- Deque supports element insertion and removal at both ends.
- The name deque is short for “double-ended queue” and is usually pronounced “deck.”

- The Deque interface extends Queue with additional methods for inserting and removing elements from both ends of the queue.

- The methods *addFirst(e)*, *removeFirst()*, *addLast(e)*, *removeLast()*, *getFirst()*, and *getLast()* are defined in the Deque interface.

CODERINDEED

Deque Methods

Type of Operation	First Element (Beginning of the Deque instance)	Last Element (End of the Deque instance)
Insert	addFirst(e) offerFirst(e)	addLast(e) offerLast(e)
Remove	removeFirst() pollFirst()	removeLast() pollLast()
Examine	getFirst() peekFirst()	getLast() peekLast()

Insert Methods

- The addfirst and offerFirst methods insert elements at the beginning of the Deque instance.
- The methods addLast and offerLast insert elements at the end of the Deque instance.
- When the capacity of the Deque instance is restricted, the preferred methods are offerFirst and offerLast because addFirst might fail to throw an exception if it is full.

Remove Methods

- removeFirst and pollFirst methods remove elements from the beginning of the Deque.
- The removeLast and pollLast methods remove elements from the end.
- The methods pollFirst and pollLast return null if the Deque is empty whereas the methods removeFirst and removeLast throw an exception if the Deque instance is empty.

Retrieve Methods

- getFirst and peekFirst retrieve the first element but don't remove the value from the Deque.
- Similarly, getLast and peekLast retrieve the last element.
- The methods getFirst and getLast throw an exception if the deque instance is empty whereas the methods peekFirst and peekLast return NULL.

Deque Implementation

The ArrayDeque class is the resizable array implementation of the Deque interface, whereas the LinkedList class is the list implementation.

- The LinkedList is more flexible than the ArrayDeque as null elements are allowed in the LinkedList implementation but not in ArrayDeque.
- ArrayDeque is more efficient than the LinkedList for add and remove operation at both ends.
- The best operation in a LinkedList implementation is removing the current element during the iteration.

ArrayDeque Constructors

ArrayDeque()

Constructs an empty array deque with an initial capacity sufficient to hold 16 elements.

ArrayDeque(Collection<? extends E> c)

Constructs a deque containing the elements of the specified collection, in the order they are returned by the collection's iterator.

ArrayDeque(int numElements)

Constructs an empty array deque with an initial capacity sufficient to hold the specified number of elements.

LinkedList Constructors

LinkedList()

Constructs an empty LinkedList.

LinkedList(Collection<? extends E> c)

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

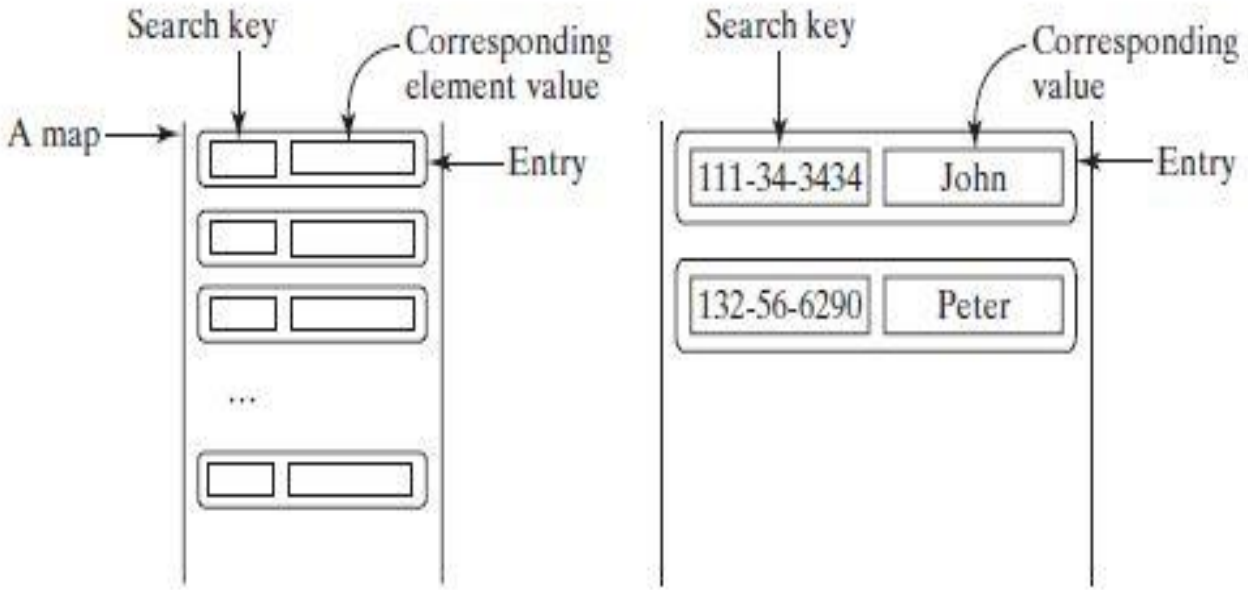
MAP INTERFACE

MAP Interface

- A map is a container object that stores a collection of key/value pairs.
- Keys are like indexes in List but in Map, the keys can be any objects.
- A map cannot contain duplicate keys.
- Each key maps to one value.
- A key and its corresponding value form an entry stored in a map.

CODERINDEED

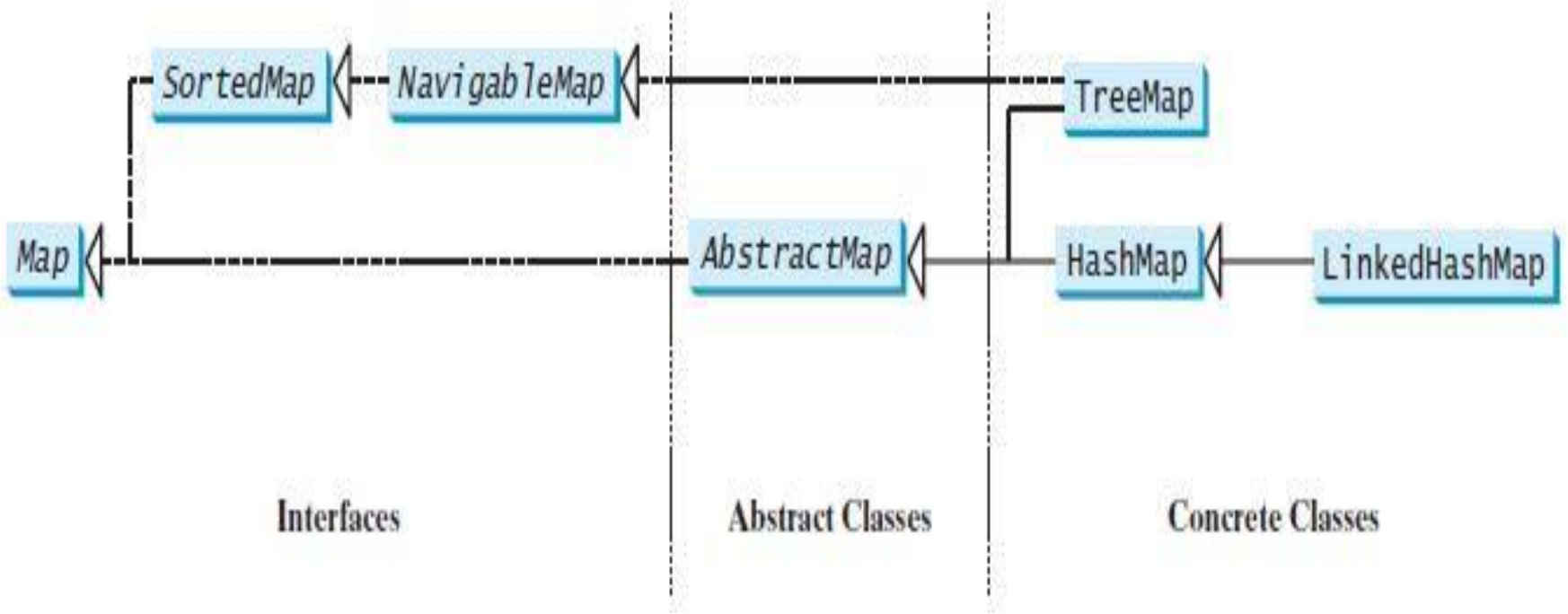
MAP

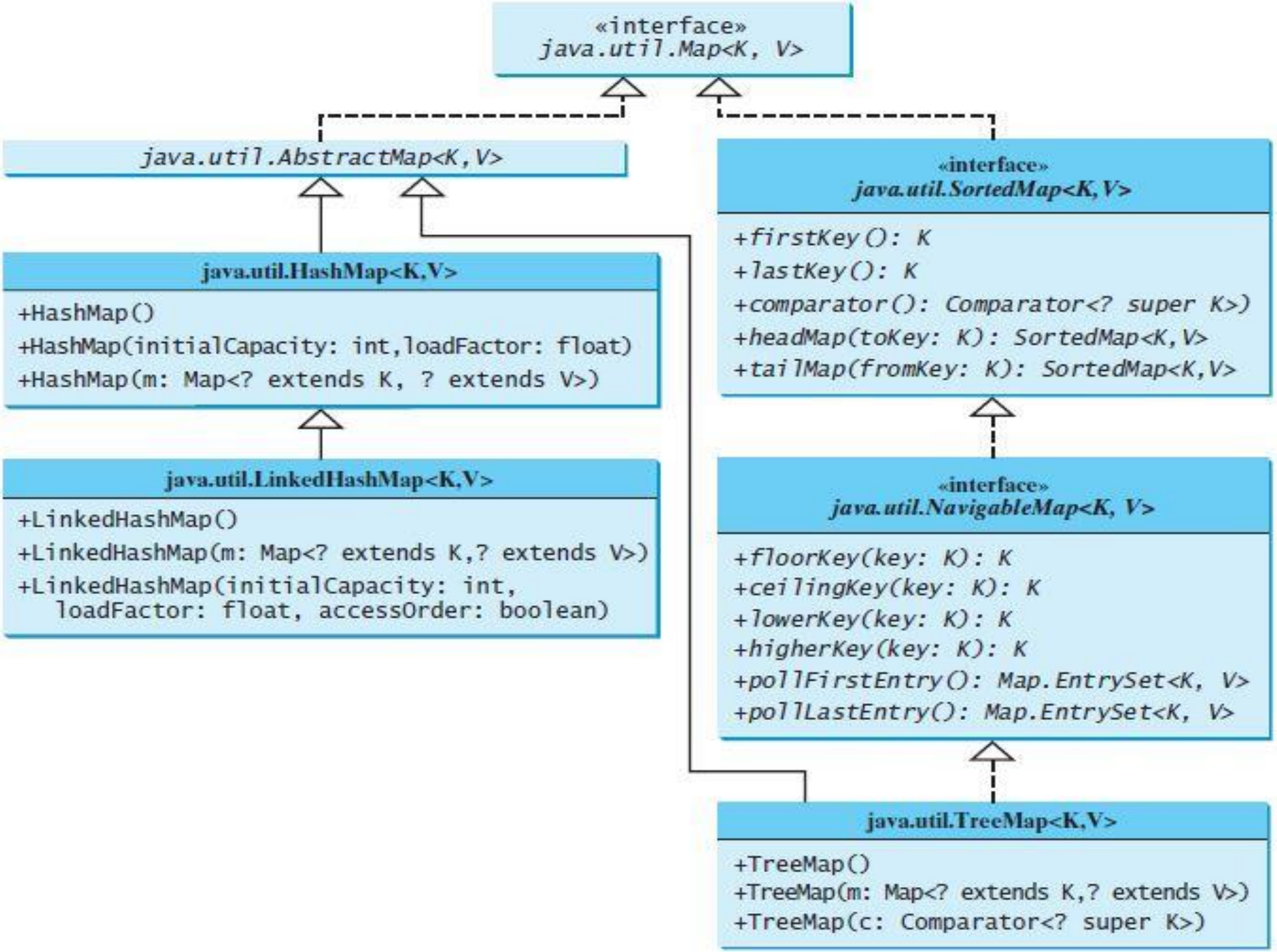


MAP Methods

- int size()
- void clear()
- boolean isEmpty()
- V put(K key, V value)
- void putAll(m: Map<? extends K,? extendsV>)
- boolean containsKey (Object key)
- boolean containsValue(Object value)
- V get(Object key)
- Set<K> keySet()
- V remove(Object key)
- Collection <V> values()
- Set<Map.Entry<K,V>> entrySet()

MAP Class Hierarchy




«interface»
java.util.Map<K, V>
java.util.AbstractMap<K,V>
«interface»
java.util.SortedMap<K,V>
+firstKey(): K
+lastKey(): K
+comparator(): Comparator<? super K>)
+headMap(toKey: K): SortedMap<K,V>
+tailMap(fromKey: K): SortedMap<K,V>
java.util.HashMap<K,V>
+HashMap()
+HashMap(initialCapacity: int,loadFactor: float)
+HashMap(m: Map<? extends K, ? extends V>)
«interface»
java.util.NavigableMap<K, V>
+floorKey(key: K): K
+ceilingKey(key: K): K
+lowerKey(key: K): K
+higherKey(key: K): K
+pollFirstEntry(): Map.EntrySet<K, V>
+pollLastEntry(): Map.EntrySet<K, V>
java.util.LinkedHashMap<K,V>
+LinkedHashMap()
+LinkedHashMap(m: Map<? extends K,? extends V>)
+LinkedHashMap(initialCapacity: int,
loadFactor: float, accessOrder: boolean)
java.util.TreeMap<K,V>
+TreeMap()
+TreeMap(m: Map<? extends K,? extends V>)
+TreeMap(c: Comparator<? super K>)

HashMap

- The HashMap class is efficient for locating a value, inserting an entry, and deleting an entry.

- The entries in a HashMap are not ordered.

public class HashMap<K, V> extends AbstractMap<K, V> implements Map<K, V>, Cloneable, Serializable

HashMap

Constructors:

- HashMap()

  Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).

- HashMap(int initialCapacity)

  Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).

- HashMap(int initialCapacity, float loadFactor)

  Constructs an empty HashMap with the specified initial capacity and load factor.

- HashMap(Map<? extends K,? extends V> m)

  Constructs a new HashMap with the same mappings as the specified Map.

Let's Do It?

Given an array containing Integers.

WAP using appropriate collection to find out the numbers which occur exactly N times.

Example: arr = {1, 3, 4, 3, 1, 1, 2, 3, 2, 4, 5};

N = 2;

Output: 4, 2

Let's Do It?

WAP to implement a method which accepts an arraylist containing duplicate values and returns a Collection having unique elements only.

public Collection revoveDuplicate(ArrayList<T extends Number> al)

Let's Do It?

WAP to create a class Student with attributes name and cgpa. Add at least 5 Student objects in an appropriate collection such that they are sorted in descending order on the basis of their cgpa.

Display the names of all the students who are eligible for Verizon placement drive. (Eligibility: cgpa >=7.5)

Let's Do It?

WAP to create a class Student with attributes name roll_no and cgpa. Add at least 5 Student objects in an appropriate collection such that they can be efficiently searched using the roll number as the key.

Display the names of all the students who are eligible for Verizon placement drive. (Eligibility: cgpa >=7.5)

Let's Do It?

WAP to create a class Car with private attributes name, mileage and price and a parameterized constructor to initialize them. Provide the getter methods for obtaining the individual attribute values.

Create 5 Car objects and store them in a collection such that the Car with best mileage can be figured out with a given price range.

Prompt the user to enter the amount within which he wants to buy the car. Display all the options available with him along with the mileage in descending order.

Let's Do It?

Given a class Employee with attributes name and salary.

Use an appropriate collection such that when a user enters the salary, it must display all the employees with given salary.