



Java Technologies

Session 1:

Learning Objectives

By the end of this session, you must be able to

- **Introduction to Java**
- **List and explain Java features**
- **Differentiate between C++ and Java**
- **Write a simple Java program**
- **HelloWorld**

Introduction to Java



Programming Language Levels

- A *programming language* specifies the words and symbols that we can use to write a program by following certain rules
- There are three programming language levels:
 - Machine language
 - Assembly language
 - High-level language
- Each type of CPU has its own specific *machine language*
- The other levels were created to make it easier for a human being to write programs

Programming Languages

- Machine language
 - computer's native language
 - sequence of zeroes and ones (binary)
 - different computers understand different sequences
 - hard for humans to understand:
 - 01010001...

Programming Languages

- Assembly language
 - mnemonics for machine language
 - *low level*: each instruction is minimal
 - still hard for humans to understand:
 - ADD, LOAD, JMP etc.,

Programming Languages

- High-level languages
 - C, C++, Java, C#, Python, JavaScript, Ruby, Perl, etc.
 - *high level*: each instruction is composed of many low-level instructions
 - closer to English and high school algebra
 - easier to read and understand
 - `hypotenuse = Math.sqrt(leg1 * leg1 + leg2 * leg2);`

Language Translators

- Machine language is the only language capable of directly instructing the CPU
- Every non machine language program instruction must be translated into machine language prior to execution
- Language Translators convert **High-level code into machine language**
- **Interpreters** translate one program statement at a time, as the program is running
- **Compilers** translate a complete program into machine language, then the machine language program is executed as needed
- Because **compiled programs run faster** than programs that are translated line by line by **an interpreter**, programmers usually choose compilers to translate frequently run business programs

Java History

- Computer language innovation and development occurs for two fundamental reasons:
 - 1) To adapt to the changing environments and uses
 - 2) To implement the refinements and improvements in the art of programming
- The development of Java was driven by both in equal measures
- Many Java features are inherited from the earlier languages:

C → C++ → Java

Before Java: C

Designed by Dennis Ritchie in 1972.

Before C:

BASIC, COBOL, FORTRAN, PASCAL

C is structured, efficient, high-level language that could replace assembly code when creating systems programs.

Before Java: C++

- Designed by Bjarne Stroustrup in 1979.
- OOP – a methodology that helps to organize complex programs through the use of polymorphism and inheritance, encapsulation.
- C++ extends C by adding object-oriented features.

Java: History

- In 1990, Sun Microsystems started a project called Green.
- **Objective:** To develop software for Consumer Electronics.
- Project was assigned to James Gosling, a veteran of classic network software design. Others included Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan.
- The team started ***writing programs in C++*** for embedding into
 - Set top boxes
 - Washing machines
 - VCR's
 - Ovens, etc.
- Aim was to make these appliances more "intelligent".

Java: History

- C++ is powerful, but also dangerous
- The power and popularity of C derived from the extensive use of pointers
- Incorrect use of pointers can cause **memory leaks**, leading the program to crash
- Replacing pointers by references, and automating memory management was the proposed solution

Java: History

- Hence, the team built a new programming language called **Oak**, which avoided potentially dangerous constructs in C++, such as **pointers, pointer arithmetic, operator overloading etc.**
- Introduced **automatic memory management**, freeing the programmer to concentrate on other things.
- **Architecture neutrality** (Platform independence)
 - Many different CPU's are used as controllers in consumer electronic devices (They may change as per the new trends in technology).
 - So, the software and programming language had to be architecture neutral.

Java: History

- It was soon realized that these design goals of consumer electronics perfectly suited an ideal programming language for the *Internet* and *WWW*, which should be:
 - ❖ Object-oriented (& support GUI)
 - ❖ Robust
 - ❖ Architecture neutral
- Internet programming presented a BIG business opportunity. Much bigger than programming for consumer electronics.
- Java was “re-targeted” for the Internet
- The team was expanded to include Bill Joy (developer of Unix), Arthur van Hoff, Jonathan Payne, Frank Yellin, Tim Lindholm etc.
- In 1994, an early web browser called **WebRunner** was written in Oak. WebRunner was later renamed **HotJava**.
- In 1995, Oak was renamed Java.
A common story is that the name Java relates to the place from where the development team got its **coffee**. The name Java survived the trade mark search.

The Java Buzzwords

- The key considerations were summed up by the Java team in the following list of buzzwords:
 - ❖ Simple
 - ❖ Secure
 - ❖ Portable
 - ❖ Object-oriented
 - ❖ Robust
 - ❖ Multithreaded
 - ❖ Architecture-neutral
 - ❖ Interpreted
 - ❖ High performance
 - ❖ Distributed
 - ❖ Dynamic

The Java Buzzwords

- **Simple** – Java is designed to be easy for the professional programmer to learn and use.
- **Object-oriented**: A clean, usable, pragmatic approach to objects
- **Robust**: Restricts the programmer to find the mistakes early, performs compile-time (strong typing) and run-time (exception-handling) checks, manages memory automatically.
- **Multithreaded**: Supports multi-threaded programming for writing program that perform concurrent computations
- **Architecture-neutral and Portable**: Java Virtual Machine provides a platform independent environment for the execution of Java byte code and can be ported to any machine.

The Java Buzzwords

Interpreted and High-performance: Java programs are compiled into an intermediate representation – byte code:

- a) Later interpreted by any JVM
- b) Translated into the native machine code (**JIT**)

Distributed: Java handles TCP/IP protocols, accessing a resource through its URL much like accessing a local file

Dynamic: Substantial amounts of run-time type information to verify and resolve access to objects at run-time.

Secure: Programs are confined to the Java execution environment and cannot access other parts of the computer.

The Java Platform

- A **platform** is the hardware and software environment in which a program runs.
- The Java platform differs from most other platforms. It's a **software-only platform** that runs on top of other, hardware-based platforms.
- The Java platform has two components:
 - **The Java Virtual Machine (Java VM)**
 - **The Java Application Programming Interface (Java API)**
 - The Java API is a large collection of ready-made software components that provide many useful capabilities, such as graphical user interface (GUI) widgets.
 - The Java API is grouped into libraries (packages) of related components which allow you to do various things.

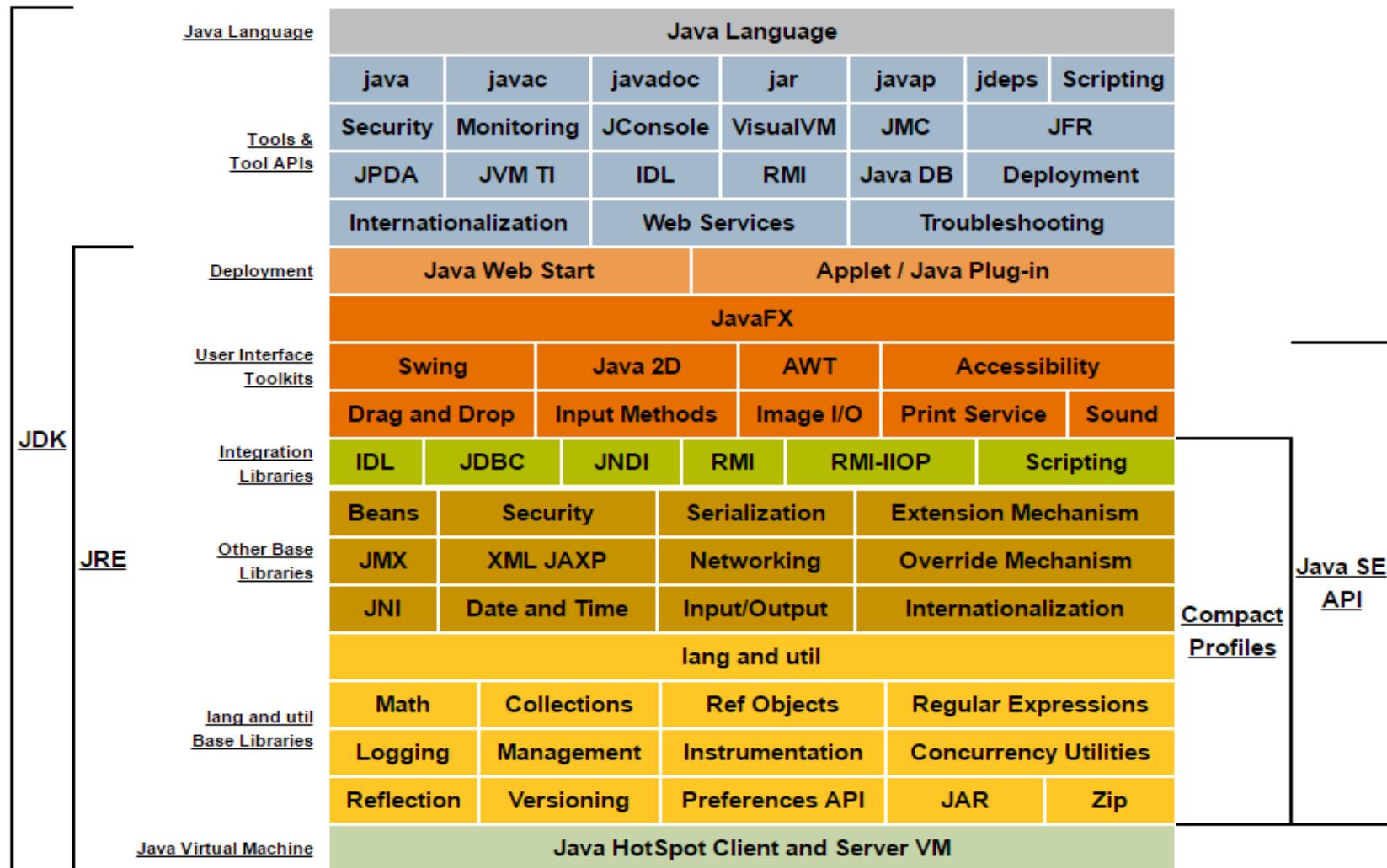
Versions of Java

- **Java Language vs. Java Platform**
 - Current version of the *language* is **18** (JDK 18 / Java SE 18)
 - Three **versions of the Java Platform**, targeted at different uses
- **Java Standard Edition (Java SE)**
 - To develop secure, portable, high-performance applications for the widest range of computing platforms possible
- **Java Enterprise Edition (Java EE)**
 - For business applications, web services, mission-critical systems
 - Transaction processing, databases, distribution, replication
- **Java Micro Edition (Java ME)**
 - Very small Java environment for smart cards, mobile phones, and set-top boxes
 - Subset of the standard Java libraries aimed at limited size and processing power

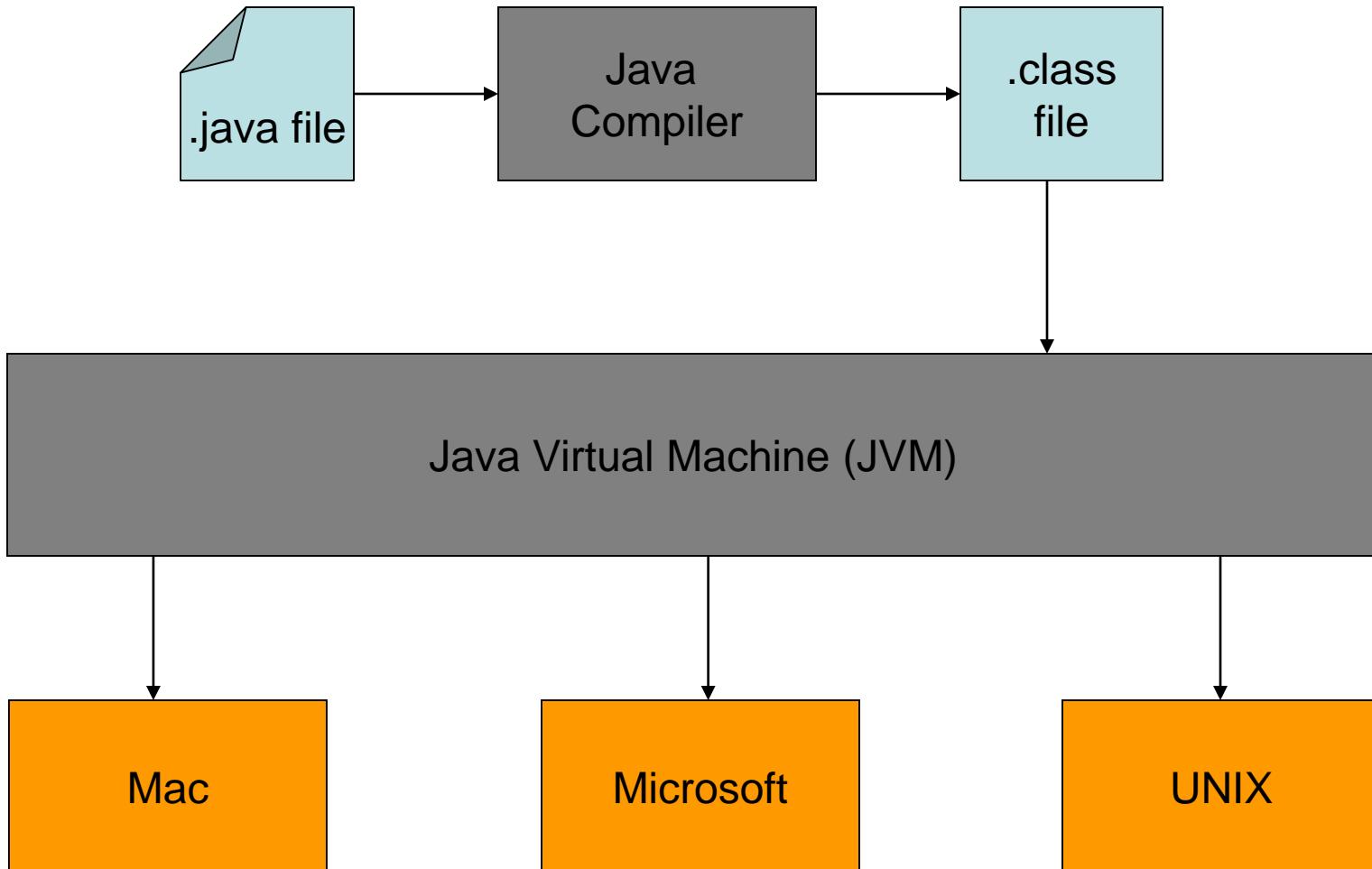
The Java SE Platform

To develop and deploy Java applications on desktops and servers and embedded environments..

Description of Java Conceptual Diagram

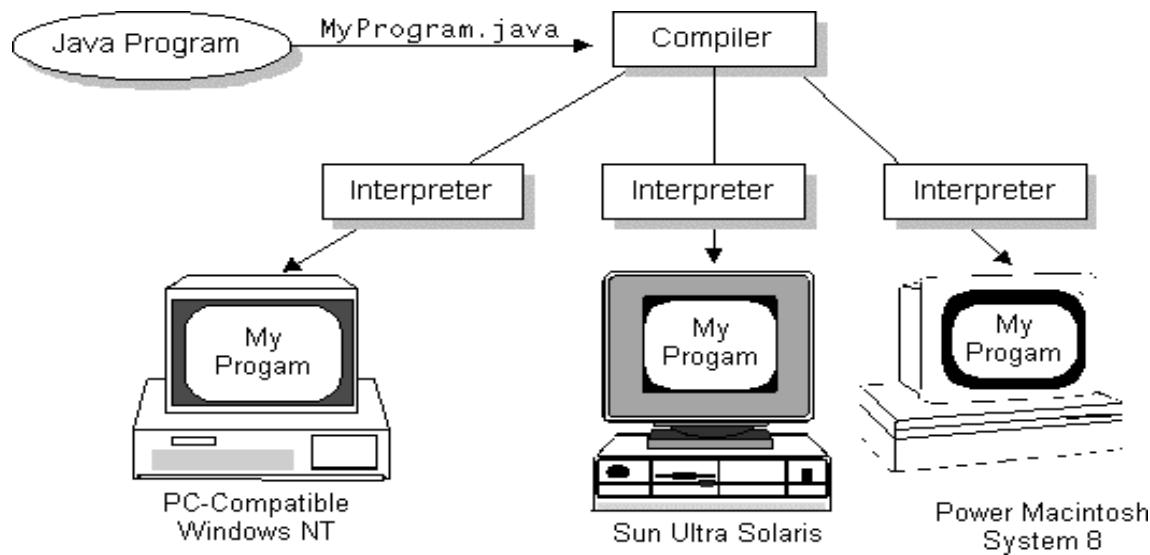


Introduction - Java Virtual Machine



Java Virtual Machine - JVM

1. Every Java interpreter, whether it's a Java development tool or a Web browser that can run Java applets, is an implementation of the Java VM
2. The Java VM can also be implemented in hardware
3. Java byte codes help make "write once, run anywhere" possible
4. We can compile Java program into byte codes on any platform that has a Java compiler
5. The byte codes can then be run on any implementation of the Java VM



Object Oriented Languages - A Comparison

	C++	Ada	Java
Encapsulation	Yes	Yes	Yes
Inheritance	Yes	No	Yes
Multiple Inheritance	Yes	No	No
Polymorphism	Yes	Yes	Yes
Binding (Early/Late)	Both	Early	Late
Concurrency	Poor	Difficult	Yes
Garbage Collection	No	No	Yes
Class Libraries	Yes	Limited	Yes

C++: OOL ADA: OBL JAVA: Almost Pure OOL

Java better than C++ ?

- No Typedef, #Defines or Preprocessor
- No Global Variables
- No goto statements
- No Pointers and Pointer arithmetic
- No Multiple Inheritance
- No Operator Overloading
- No copy constructors, destructors
- No Templates

Added or Improved over C++

- Interfaces
- Automatic Garbage collection
- Exceptions (More powerful than C++)
- Strings
- Packages
- Multi-threading
- instanceof

Types of Java Applications

- Different ways to write/run a Java codes are:

Application- A stand-alone program that can be invoked from command line . A program that has a “main()” method

Applet- A program embedded in a web page , to be run when the page is browsed .

A program that contains no “main” method

- Application - Java interpreter
- Applets - Java enabled web browser (Linked to HTML via <APPLET> tag in .html file)

Java Program Structure

- The Java programming language:
 - A program is made up of one or more **classes**
 - A class contains **data**
 - and One or more **methods**
- A Java application always contains a method called **main()**

Java Program Structure

```
// comments about the class
```

```
public class MyProgram
```

```
{
```

class body

```
}
```

class header

Comments can be added almost anywhere

Java Program Structure

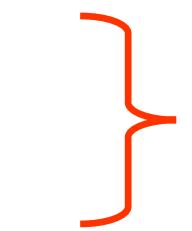
```
// comments about the class

public class MyProgram
{
    // Data members

    // comments about the method
    public static void main (String[] args)
    {
        }
    }

}
```

method header 

method body 

Comments

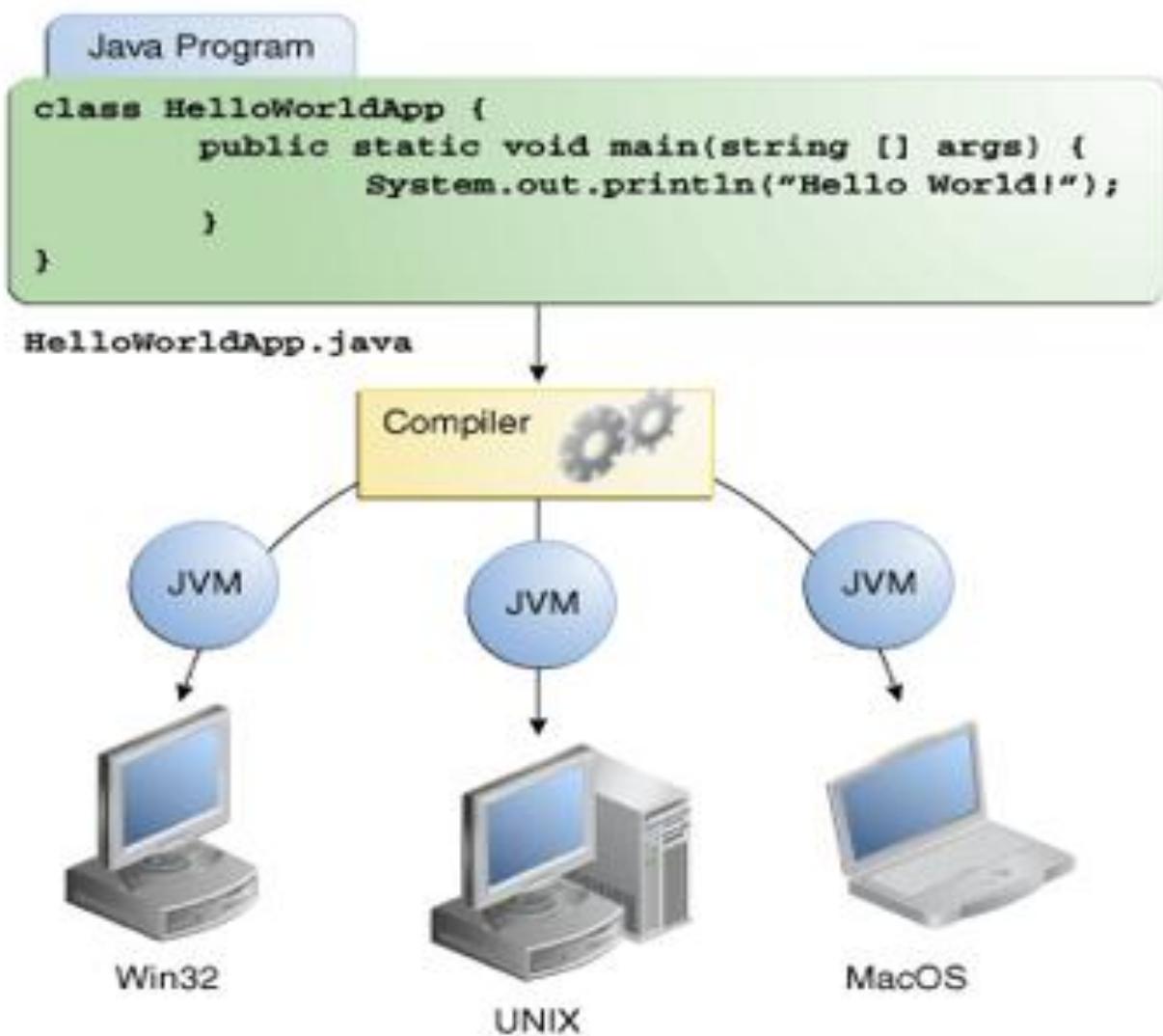
- Comments in a program are also called *inline documentation*
- They should be included to explain the purpose of the program and describe processing steps
- They do not affect how a program works
- Java comments can take two forms:

```
// this comment runs to the end of the line
```

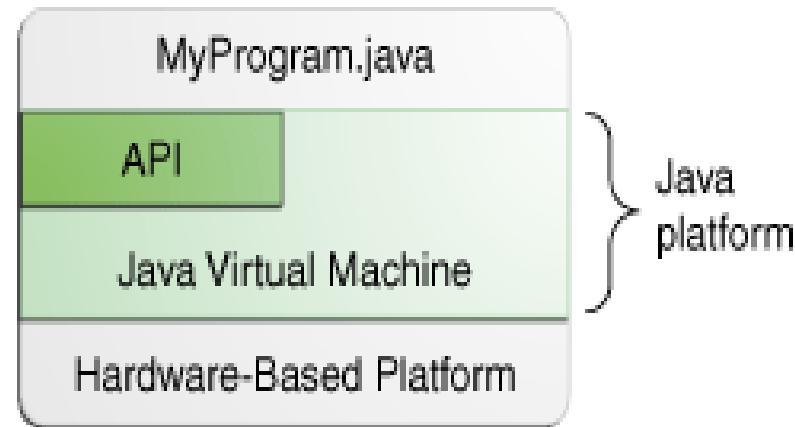
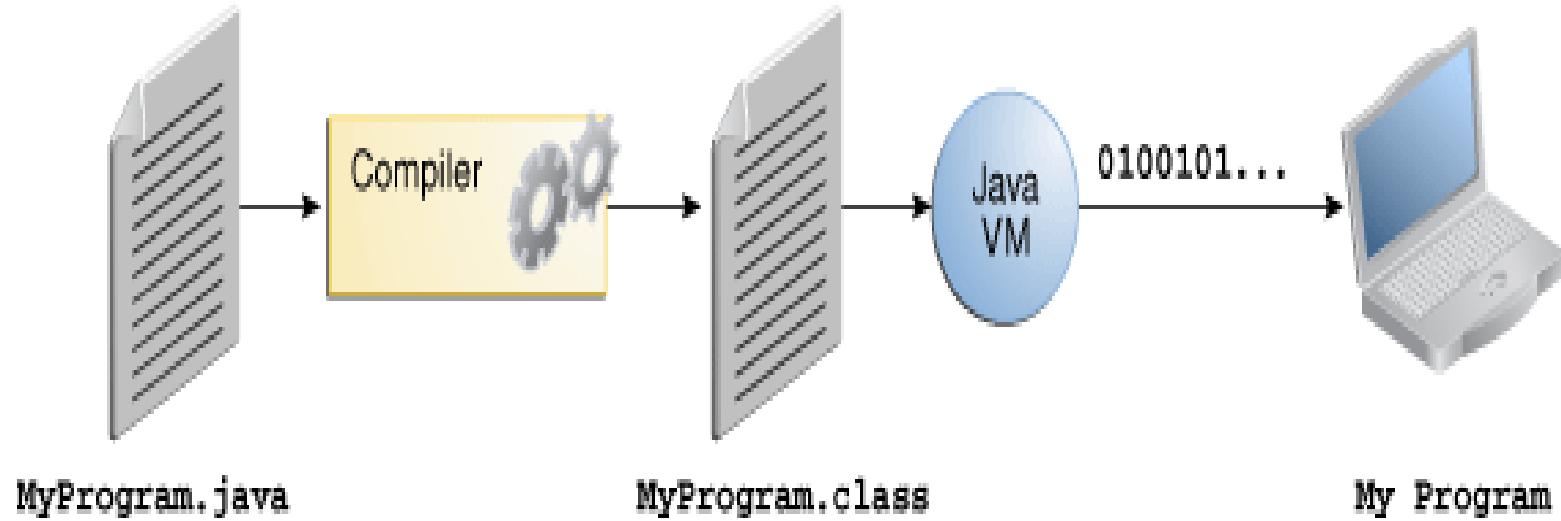
```
/* this comment runs to the terminating
symbol, even across line breaks */
```

```
/** This kind of comment is a special 'javadoc' style comment */
```

Java Program - Example



Java Program - Example



A Simple Java Application

- Define a class HelloWorld and store it into a file: HelloWorld.java

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

Compile the program: `javac HelloWorld.java`

Execute the program: `java HelloWorld`

Output: Hello World

Try Yourself : [HelloWorld.java](#)

A Simple Java Application

```
public class HelloWorld1{  
    public void display()  
    {  
        System.out.println("Hello World");  
    }  
    public static void main(String[] args)  
    {  
        HelloWorld1 hw = new HelloWorld1();  
        hw.display();  
    }  
}
```

Compile the program: `javac HelloWorld1.java`

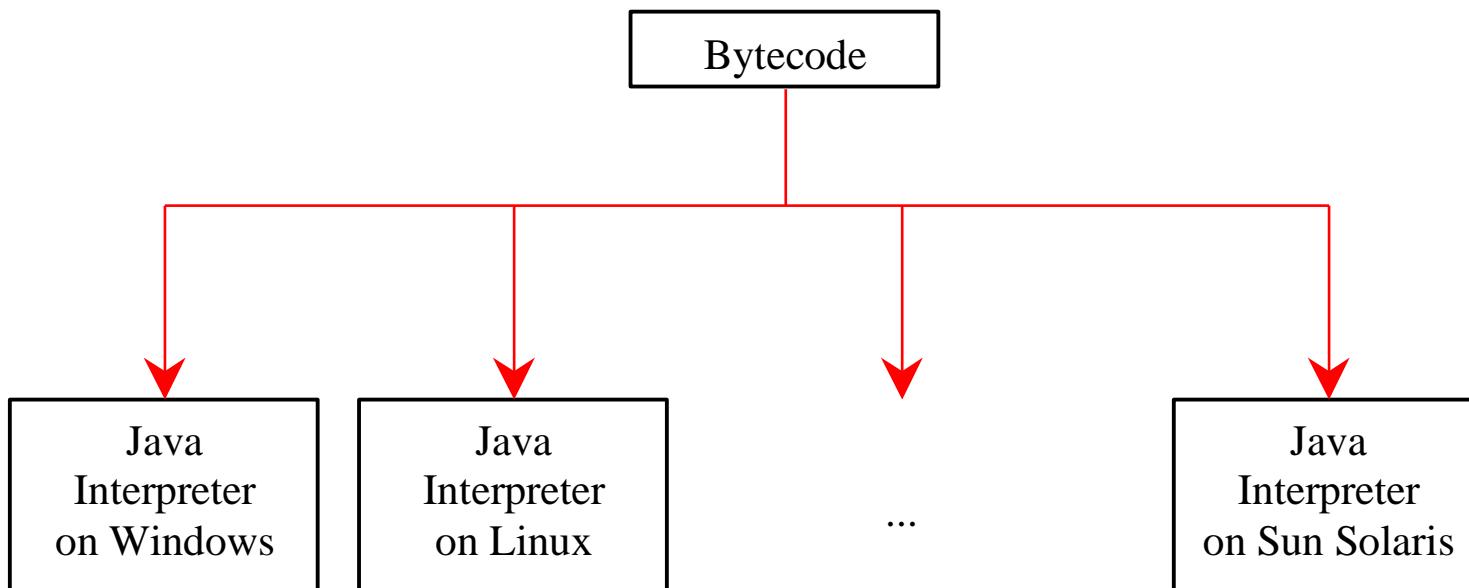
Execute the program: `java HelloWorld`

Output: Hello World

Try Yourself: [HelloWorld1.java](#)

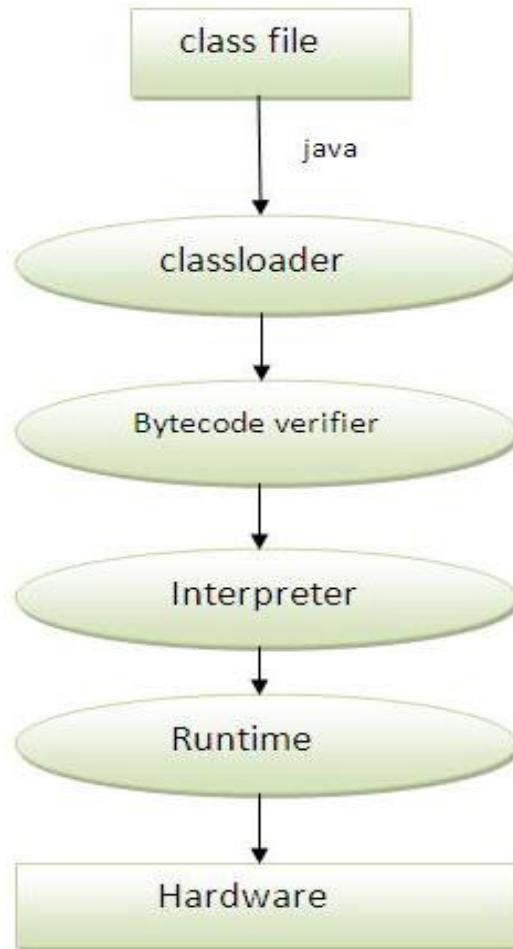
Executing Applications

- On command line
 `java classname`

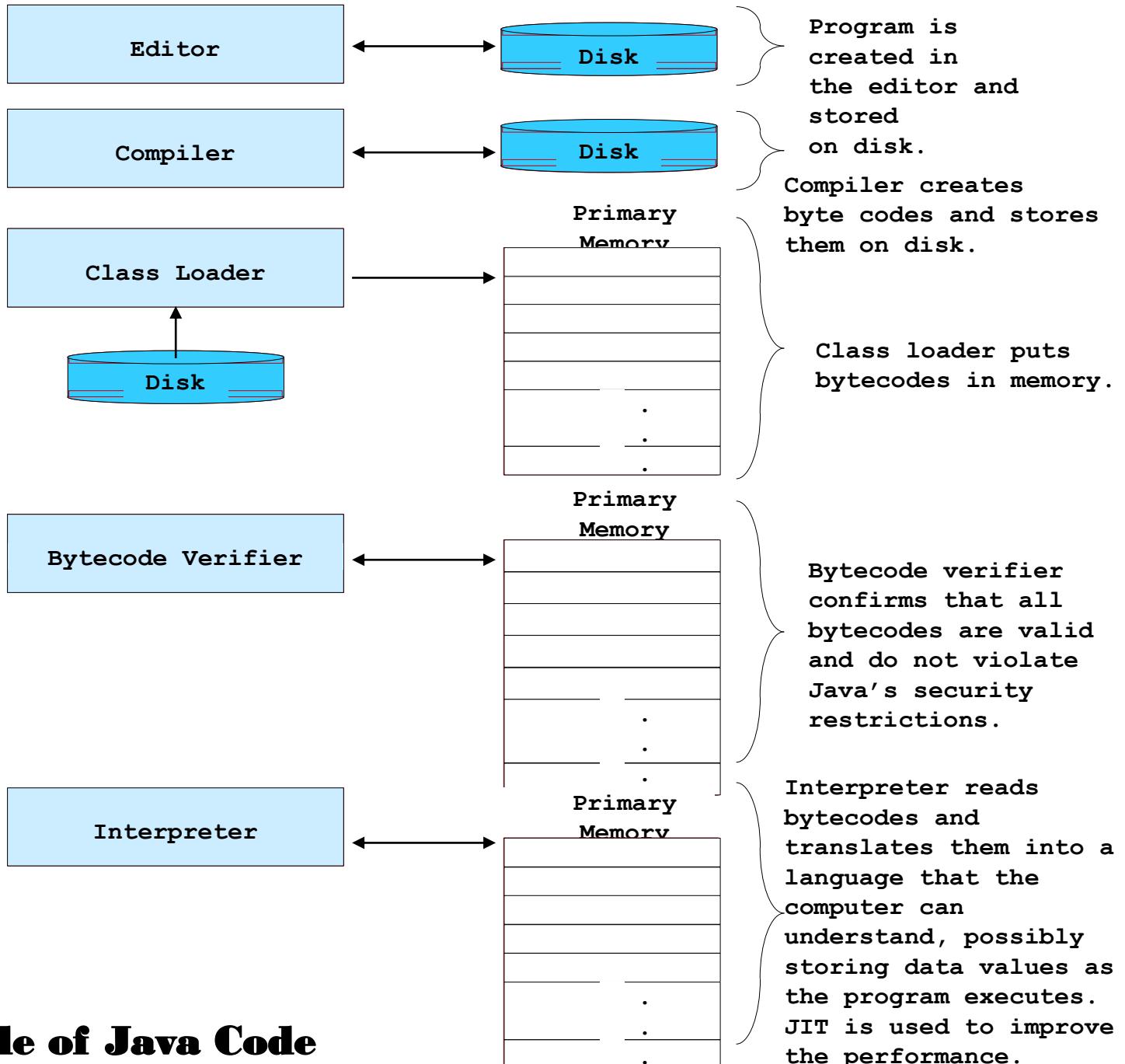


Java Program - Example

At runtime, following steps are performed:



Phase 1



Command Line Argument

The command-line argument is an argument passed at the time of running the java program. The argument can be received in the program and used as an input. So, it provides an convenient way to check out the behaviour of the program on different values. You can pass **N** numbers of arguments from the command prompt.

Simple example of command-line argument

In this example, we are receiving only one argument and printing it. For running this program, you must pass at least one argument from the command prompt.

```
class A{
public static void main(String args[]){
    System.out.println("first argument is: "+args[0]);
}
```

Compile: javac A.java

Run: java A cdachyd

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

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

Next

- **Basic programming constructs of Java**
- **Classes and Objects in Java**
- **new operator**
- **Constructors**
- **Overloading**



Session 2:

Learning Objectives

By the end of this session, you must be able to

- **Explain basic programming constructs of Java**
- **Arrays in Java**
- **Explain Classes and Objects in Java**
- **Use instance data and methods**
- **Use *new* operator to create instances**
- **Explain Constructors - Overloading**
- **Explain Method overloading**
- **Write programs on Parameter passing – object as parameter**

Java Programming - Basic Constructs

Char Set

16 – bit Unicode char set

Identifiers:

- ***Identifiers*** are the named words a programmer uses in a program
- An identifier can be made up of letters, digits, the underscore character (_), and the dollar sign (\$)
- They cannot begin with a digit
- Java is *case sensitive*, therefore Total and total are different identifiers

Reserved Words

- Often we use special identifiers called *reserved words* that already have a predefined meaning in the language
- A reserved word cannot be used in any other way

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

Variables

- Programming languages uses variables to store data
- To allocate memory space for a variable, JVM requires:
 - 1) To specify the data type of the variable
 - 2) To associate an identifier with the variable
 - 3) Optionally, the variable may be assigned an initial value
- All done as part of variable declaration.

Basic Variable Declaration

- Syntax:
datatype identifier [=value];
- Datatype must be
 - A simple data type
 - User defined datatype (Class type)
- Value is an optional initial value.

We can declare several variables at the same time:

type identifier [=value][, identifier [=value] ...];

Examples:

```
int a, b, c;  
int d = 3, e, f = 5;  
byte g = 22;  
double pi = 3.14159;  
char ch = 'x';
```

Constants

- A constant is an identifier that is similar to a variable except that it holds one value for its entire existence
- The compiler will issue an error if you try to change a constant
- In Java, we use the **final** modifier to declare a constant

Example: **final int MIN_HEIGHT = 69;**

Data Types - Primitive

Java defines eight simple (primitive) types:

1. byte – 8-bit integer type
2. short – 16-bit integer type
3. int – 32-bit integer type
4. long – 64-bit integer type
5. **float – 32-bit floating-point type**
6. **double – 64-bit floating-point type**
7. **char – symbols in a character set (16-bit Unicode)**
8. boolean – logical values true and false

Introduction - Data Types

Data type	Bytes	Min Value	Max Value	Literal Values
byte	1	-2^7	$2^7 - 1$	123
short	2	-2^{15}	$2^{15} - 1$	1234
int	4	-2^{31}	$2^{31} - 1$	12345, 086, 0x675
long	8	-2^{63}	$2^{63} - 1$	123456
float	4	-	-	1.0
double	8	-	-	123.86
char	2	0	$2^{16} - 1$	'a', '\n'
boolean	-	-	-	true, false

General rule:

Min value = $2^{(\text{bits} - 1)}$

Max value = $2^{(\text{bits}-1)} - 1$
(where 1 byte = 8 bits)

Data Types

- boolean result = true;
- char capitalC = 'C';
- byte b = 100;
- short s = 10000;
- int i = 100000;
- // The number 26, in decimal
- int decVal = 26;
- // The number 26, in hexadecimal
- int hexVal = 0x1a;
- // The number 26, in binary
- int binVal = 0b11010;

```
long creditCardNumber = 1234_5678_9012_3456L;
```

```
long socialSecurityNumber = 999_99_9999L;
```

```
float pi = 3.14_15F;
```

```
long bytes = 0b11010010_01101001_10010100_10010010;
```

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Operators

- Java operators are used to build value expressions
- Java provides a rich set of operators:
 - 1) Assignment
 - 2) Arithmetic
 - 3) Relational
 - 4) Logical
 - 5) Bitwise
 - 6) Unary Operators
 - 7) Ternary operator
 - 8) Special operators

Operators - Assignment Operators/Arithmetic Operators

- Assignment Operator

Operator	Description	Example
=	Assignment	int i = 10; int j = i;

- Arithmetic Operators

Operator	Description	Example
+	Addition	int i = 8 + 9; byte b = (byte) 5+4;
-	Subtraction	int i = 9 - 4;
*	Multiplication	int i = 8 * 6;
/	Division	int i = 10 / 2;
%	Remainder	int i = 10 % 3;

[Example:ArithmeticDemo.java](#)

Operators - Unary Operators/Equality Operators

- Unary Operators

Operator	Description	Example
+	Unary plus	int i = +1;
-	Unary minus	int i = -1;
++	Increment	int j = i++;
--	Decrement	int j = i--;
!	Logical Not	boolean j = !true;

Example: [UnaryDemo.java](#)

Example: [PrePostDemo.java](#)

- Equality Operators

Operator	Description	Example
==	Equality	If (i==1)
!=	Non equality	If (i != 4)

Operators - Relational Operators/Conditional Operators

- Relational Operators

Operator	Description	Example
>	Greater than	if (x > 4)
<	Less than	if (x < 4)
>=	Greater than or equal to	if (x >= 4)
<=	Less than or equal to	if (x <= 4)

- Conditional Operators

Operator	Description	Example
&&	Conditional and	If (a == 4 && b == 5)
	Conditional or	If (a == 4 b == 5)

Example: [ConditionalAndOrDemo.java](#)

Example: [TernaryOperator.java](#)

Operators - instanceof Operator/Bitwise Operators/shift operators

- instanceof Operator

Operator	Description	Example
instanceof	Instance of	If (john instanceof person)

- Bitwise Operators

Operator	Description	Example
&	Bitwise and	001 & 111 = 1
	Bitwise or	001 110 = 111
^	Bitwise ex-or	001 ^ 110 = 111
~	Reverse	~0 = 1

Example: [BitDemo.java](#)

- Shift Operators

Operator	Description	Example
>>	Right shift	4 >> 1 = 0100 >> 1 = 0010 = 2
<<	Left Shift	4 << 1 = 0100 << 1 = 1000 = 8
>>>	Unsigned Right shift	4 >>> 1 = 0100 >>> 1 = 0010 = 2

Operators – Points Know

- **Increment & Decrement Operators:**
 - can't apply increment & decrement operators for constants ex: int x=++4;
 - can't apply increment & decrement operators for final variable ex: final int x=4; x++;
 - Can apply increment & decrement operators for any primitive type except boolean
- **Arithmetic Operators:**
 - There is no way to represent infinity in case of integral arithmetic (int, byte, short, long). Hence if infinity is result , we always get **ArithmeticException**: / by zero ex: System.out.println(10/0);
 - In case of floating point arithmetic, there is always a way to represent infinity. Float and Double classes contain the following constants:
 - **Positive_Infinity** and **Negative_Infinity** Ex: System.out.println(10/0.0); System.out.println(-10/0.0);
 - In integral arithmetic, there is no way to represent undefined results. Ex: 0/0=undefined , So leads to **ArithmeticException**
 - In floating arithmetic, undefined results are **NaN (Not a Number)** Ex: System.out.println(0/0.0); // NaN
 - The only operators which cause **ArithmeticException** are / and %
- **Relational Operators (<, >, <=, >=) :**
 - Can apply relational operators for every primitive data type except **boolean**
 - Can't be applied to reference types

Operators

- **Equality Operators (==, !=) :**
 - can apply equality operators for every primitive type including boolean types
 - Can apply even for object references also
- **instanceof Operator:**
 - By using *instanceof* operator, whether the given object is of particular type or not.
 - Example: Thread t=new Thread();
`System.out.println(t instanceof Thread); // true`
`System.out.println(t instanceof Object); // true`
`System.out.println(t instanceof Runnable); // true`
- **Bitwise Operators:**
 - & (AND) , | (OR) , ^ (XOR) , ~ (Negation), ! (NOT)
 - ~ (tilde) can't be applied to boolean types
 - ! (NOT) can't be applied to integral types
- **new Operator :**
 - Used create objects
 - No delete operator as objects destroyed automatically – Garbage Collection

Operator Precedence

Operator Precedence

Operators	Precedence
postfix	<code>expr++ expr--</code>
unary	<code>++expr --expr +expr -expr ~ !</code>
multiplicative	<code>* / %</code>
additive	<code>+ -</code>
shift	<code><< >> >>></code>
relational	<code>< > <= >= instanceof</code>
equality	<code>== !=</code>
bitwise AND	<code>&</code>
bitwise exclusive OR	<code>^</code>
bitwise inclusive OR	<code> </code>
logical AND	<code>&&</code>
logical OR	<code> </code>
ternary	<code>? :</code>
assignment	<code>= += -= *= /= %= &= ^= = <<= >>= >>>=</code>

Selection Statements

- Java selection statements allow us to control the flow of program's execution based upon conditions known only during run-time.
- Java provides the following selection statements:
 - 1) if-then
 - 2) if - then - else
 - 3) switch

If – then:

```
if (isMoving){  
// the "then" clause: decrease current speed  
currentSpeed--;  
}
```

Flow Control - if-else

Syntax	Example
<pre>if (<condition-1>) { // logic for true condition-1 goes here } else if (<condition-2>) { // logic for true condition-2 goes here } else { // if no condition is met, control comes here }</pre>	<pre>int a = 10; if (a < 10) { System.out.println("Less than 10"); } else if (a > 10) { System.out.println("Greater than 10"); } else { System.out.println("Equal to 10"); }</pre> <p>Result: Equal to 10s</p>

```
* if(true)  
System.out.println("Hello");  
** if(true)  
int x=10; //  
System.out.println("Hello");  
*** if(true) {  
    int x=10;  
}  
**** if(true);
```

Example: [IfElseDemo.java](#)

Flow Control - switch

Syntax	Example
<pre>switch (<value>) { case <a>: // stmt-1 break; case : //stmt-2 break; default: //stmt-3 }</pre>	<pre>int a = 10; switch (a) { case 1: System.out.println("1"); break; case 10: System.out.println("10"); break; default: System.out.println("None"); } Result: 10</pre>

```
* byte b=10;  
switch(b){  
}  
  
** char ch='a';  
switch(ch){  
}  
  
*** long l=10l;  
switch(l){  
} //  
  
**** boolean b=true;  
switch(b){  
}//  
  
***** String color="blue";  
switch(color){  
}
```

Example: [SwitchDemo.java](#)

Example: [SwitchDemoWithStrings.java](#)

Iteration Statements

Java iteration statements enable repeated execution of part of a program until a certain termination condition becomes true.

Java provides three iteration statements:

- 1) do - while
- 2) while
- 3) for

Flow Control - do-while / while

- **do-while**

Example: [DoWhileDemo.java](#)

Syntax	Example
<pre>do { // stmt-1 } while (<condition>);</pre>	<pre>int i = 0; do { System.out.println("In do"); i++; } while (i < 10);</pre> <p>Result: Prints “In do” 10 times</p>

- **while**

Example: [WhileDemo.java](#)

Syntax	Example
<pre>while (<condition>) { //stmt }</pre>	<pre>int i = 0; while (i < 10) { System.out.println("In while"); i++; }</pre> <p>Result: “In while” 10 times</p>

Flow Control - for loop

- for

Example: [ForDemo.java](#)

Syntax	Example
<pre>for(initialize; condition; expression) { // stmt }</pre>	<pre>for (int i = 0; i < 10; i++) { System.out.println("In for"); }</pre> <p>Result: Prints “In for” 10 times</p>

```
class ForEachDemo // for each style
{
    public static void main(String[] args) {
        int arr[]={10,20,30,40,50};
        for (int i: arr)
        {
            System.out.println("Count="+i);
        }
    }
}
```

Iteration Statements

```
*while(true)  
System.out.println("Hello");
```

```
*while(true);
```

```
*while(true)  
int x=10; //
```

```
*while(true){  
int x=10;  
}
```

```
* while(true){  
    System.out.println("Hello");  
}  
    System.out.println("Hi"); //urc
```

```
*while(false){  
    System.out.println("Hello"); //urc  
}  
    System.out.println("Hi");
```

```
* int a=10, b=20;  
while(a<b){  
    System.out.println("Hello");  
}  
    System.out.println("Hi");
```

```
* final int a=10, b=20;  
while(a<b){  
    System.out.println("Hello");  
}  
    System.out.println("Hi"); // urc
```

```
for(; ;); //true  
  
for(int i=0; true; i++)  
{  
    System.out.println("Hello");  
}  
    System.out.println("HI"); //urc
```

```
for(int i=0; false; i++)  
{  
    System.out.println("Hello");  
}  
    System.out.println("HI"); //urc
```

```
for(int i=0; ; i++)  
{  
    System.out.println("Hello");  
}  
    System.out.println("HI");  
//urc
```

Jump Statements

Java jump statements enable transfer of control to other parts of program.

Java provides three jump statements:

- 1) break
- 2) continue
- 3) return

In addition, Java supports exception handling that can also alter the control flow of a program.

break Statement

- The break statement has two forms:
 - labeled and unlabeled
- We saw the unlabeled form in the previous discussion of the switch statement.
- We can also use an unlabeled break to terminate a for, while, or do-while loop

[Example:BreakDemo.java](#)

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

Example:ContinueDemo.java

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
 - one that doesn't return.
- To return a value, simply put the value after the return keyword.

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

```
return;
```

Coding Guidelines

class name	should begin with uppercase letter and be a noun e.g.String, System, Thread etc.
Interface name	should begin with uppercase letter and be an adjective (wherever possible). e.g. Runnable, ActionListener etc.
method name	should begin with lowercase letter and be a verb. e.g. main(), print(), println(), actionPerformed() etc.
variable name	should begin with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter. e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

Arrays in JAVA

Declaring an Array Variable

- Syntax and Examples
 - **<type> [] variable_name;**
 - *int [] prime;*
 - *int prime[];*
- Both syntaxes are equivalent
- No memory allocation at this point

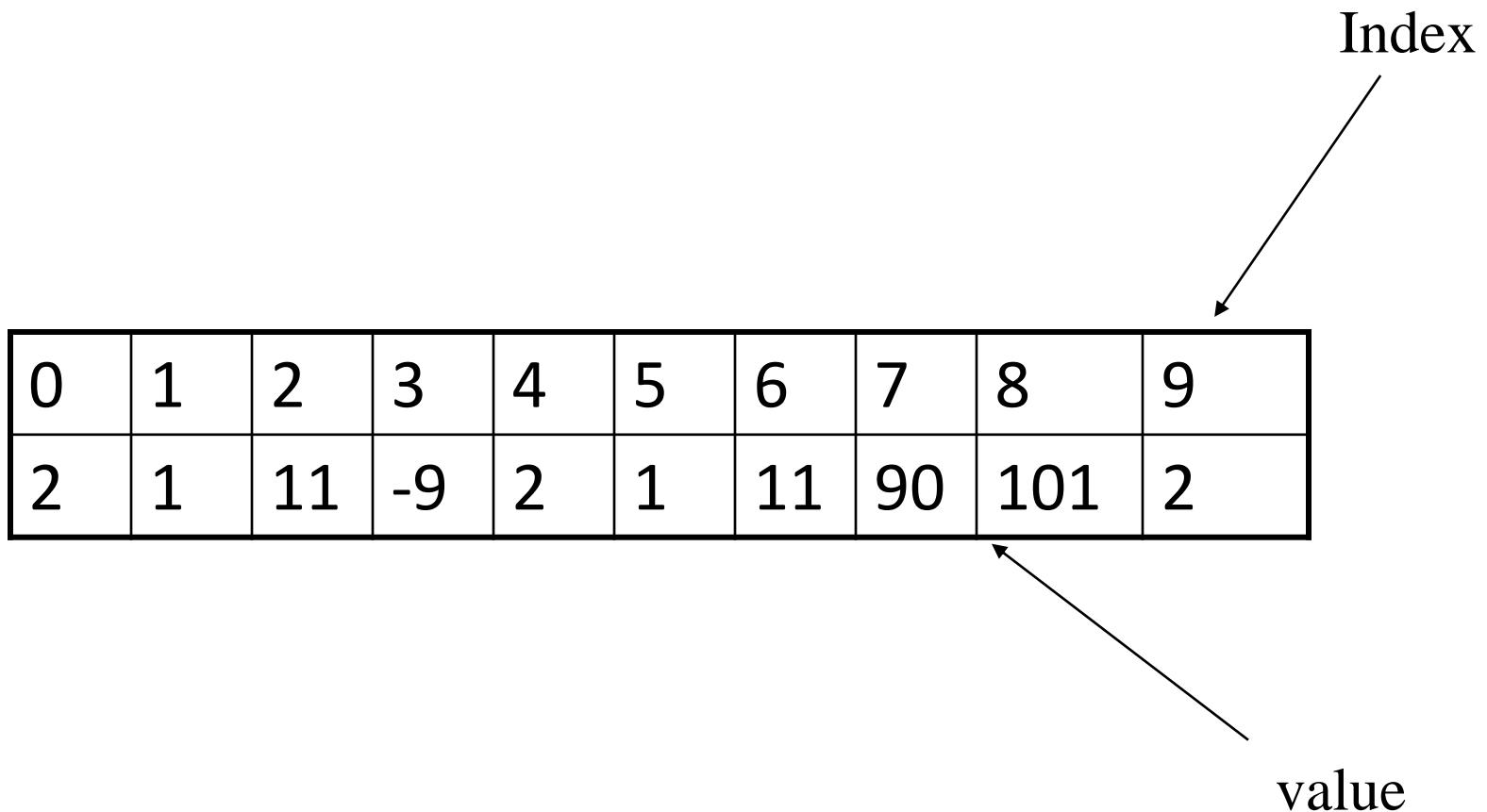
Defining an Array

- **Define an array** as follows:

```
variable_name=new <type>[N];  
primes=new int[10];
```

- **Declaring and defining** in the same statement:
int[] primes=new *int*[10];
- In JAVA, int is of 4 bytes, total space=4*10=40 bytes

Graphical Representation



What happens if ...

- We define

```
int[] prime=new long[20];
```

Primes.java:5: incompatible types

found: long[]

required: int[]

int[] primes = new long[20];

 ^

- The right hand side defines an array, and thus the array variable should refer to the same type of array

What happens if ...

- We define

int prime[100];

MorePrimes.java:5:]' expected

- The C++ style is not permitted in JAVA syntax

Default Initialization

- When array is created, array elements are initialized
 - Numeric values (int, double, etc.) to 0
 - Boolean values to `false`
 - Char values to '`\u0000`'
 - Class types to `null`

Accessing Array Elements

- Index of an array is defined as
 - **Positive int, byte or short values**
 - Expression that results into these types
- Any other types used for index will give error
 - long, double, etc.
 - Incase expression results in long, then type cast to int
- Indexing starts from 0 and ends at N-1

```
primes[2]=0;  
int k = primes[2];  
...
```

Validating Indexes

- JAVA checks whether the index values are valid at runtime
 - If **index is negative** or greater than the size of the array then an **IndexOutOfBoundsException** will be thrown
 - Program will normally be terminated unless handled in the try {} catch {}

What happens if ...

```
long[] primes = new long[20];  
primes[25]=33;
```

....

Runtime Error:

Exception in thread “main”
java.lang.ArrayIndexOutOfBoundsException: 25
at Primes.main(Primes.java:6)

Initializing Arrays

- Initialize and specify size of array while declaring an array variable

```
int[] primes={2,3,5,7,11,13,17}; //7 elements
```

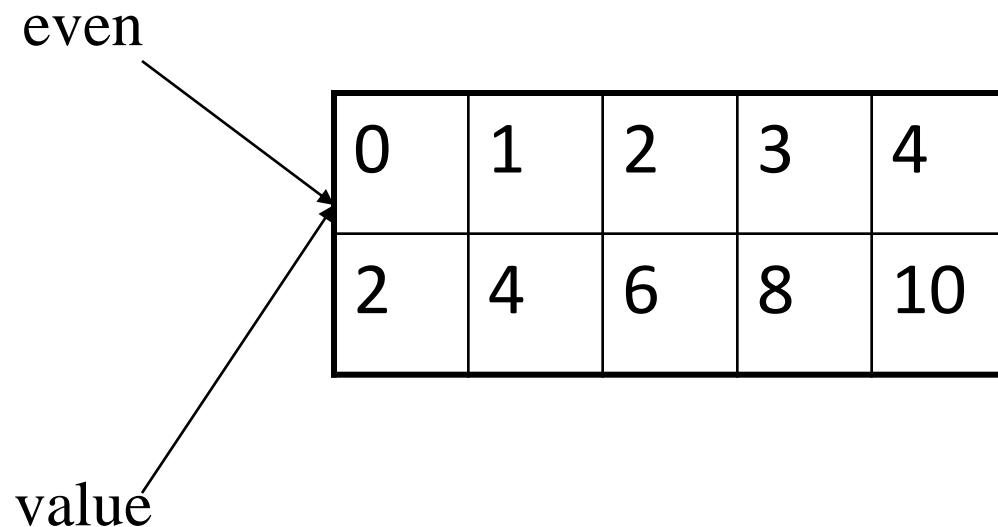
- You can initialize array with an existing array

```
int[] even={2,4,6,8,10};
```

```
int[] value=even;
```

- One array but two array variables!
- Both array variables refer to the same array
- Array can be accessed through either variable name

Graphical Representation



[ArrayOne.java](#)

Array Length

- Refer to array length using *length*
 - A data member of array object
 - array_variable_name.length
 - for(int k=0; k<primes.length;k++)
....
- Sample Code:

```
long[] primes = new long[20];
System.out.println(primes.length);
```
- Output: 20
- If number of elements in the array are changed, JAVA will automatically change the length attribute!

Sample Program

```
class MinAlgorithm
{
    public static void main ( String[] args )
    {
        int[] array = { -20, 19, 1, 5, -1, 27, 19, 5 } ;
        int min=array[0]; // initialize the current minimum

        for ( int index=0; index < array.length; index++ )

            if ( array[ index ] < min )
                min = array[ index ] ;

        System.out.println("The minimum of this array is: " + min );
    }
}
```

Arrays of Arrays

- Two-Dimensional arrays

```
float[][] temperature=new float[10][365];
```

- 10 arrays each having 365 elements
- First index: specifies array (row)
- Second Index: specifies element in that array (column)
- In JAVA float is 4 bytes, total Size= $4 * 10 * 365 = 14,600$ bytes

Initializing Array of Arrays

```
int[][] array2D = {  
    {99, 42, 74, 83, 100},  
    {90, 91, 72, 88, 95},  
    {88, 61, 74, 89, 96},  
    {61, 89, 82, 98, 93},  
    {93, 73, 75, 78, 99},  
    {50, 65, 92, 87, 94},  
    {43, 98, 78, 56, 99} };  
  
//7 arrays with 5 elements each
```

Arrays of Arrays of Varying Length

- All arrays do not have to be of the same length

```
float[][] samples;
```

```
samples=new float[6][];//defines # of arrays
```

```
samples[2]=new float[6];
```

```
samples[5]=new float[101];
```

- Not required to define all arrays

Initializing Varying Size Arrays

```
int[][] uneven = {{ 1, 9, 4 }, { 0, 2 }, { 0, 1, 2, 3, 4 } };  
  
//Three arrays  
  
//First array has 3 elements  
//Second array has 2 elements  
//Third array has 5 elements
```

Array of Arrays Length

```
long[][] primes = new long[20][];
primes[2] = new long[30];
System.out.println(primes.length); //Number of arrays
System.out.println(primes[2].length);//Number of elements
in the second array
```

OUTPUT:

20

30

Sample Program

```
class unevenExample3
{
    public static void main( String[] arg )
    { // declare and construct a 2D array
        int[][] uneven = { { 1, 9, 4 }, { 0, 2 }, { 0, 1, 2, 3, 4 } }
    ;
        // print out the array
        for ( int row=0; row < uneven.length; row++ ) //changes
row
        {
            System.out.print("Row " + row + ": ");
            for ( int col=0; col < uneven[row].length; col++ ) //changes column
                System.out.print( uneven[row][col] + " " );
            System.out.println();
        }
    }
}
```

Row 0: 1 9 4

Row 1: 0 2

Row 2: 0 1 2 3 4

Multidimensional Arrays

- A farmer has 10 farms of beans each in 5 countries, and each farm has 30 fields!
- Three-dimensional array

```
long[][][] beans=new long[5][10][30];  
//beans[country][farm][fields]
```

Varying length in Multidimensional Arrays

- Same features apply to multi-dimensional arrays as those of 2 dimensional arrays

```
long beans=new long[3][][];//3 countries
```

```
beans[0]=new long[4][];//First country has 4 farms
```

```
beans[0][3]=new long[10];
```

```
// Fourth farm in first country has 10 fields
```

Coding Guidelines

class name	should begin with uppercase letter and be a noun e.g.String, System, Thread etc.
Interface name	should begin with uppercase letter and be an adjective (whereever possible). e.g. Runnable, ActionListener etc.
method name	should begin with lowercase letter and be a verb. e.g. main(),print(),println(),actionPerformed() etc.
variable name	should begin with lowercase letter e.g. firstName,orderNumber etc.
package name	should be in lowercase letter. e.g. java,lang,sql,util etc.
constants name	should be in uppercase letter. e.g. RED,YELLOW,MAX_PRIORITY etc.

What's a Program?

- Model of Complex system
 - **model**: simplified representation of salient features of something, either tangible or abstract
 - **system**: collection of collaborating components
- Programming Paradigm or Approach
 - A programming paradigm is a style or “way” of programming
 - **Example**: OOP, POP, Declarative, functional, etc.,

What's a Program?

- Sequences of instructions expressed in specific programming language
 - **syntax**: grammatical rules for forming instructions
 - **semantics**: meaning/interpretation of instruction
- Programming languages
 - **Examples**: C, C++, Java,etc.,

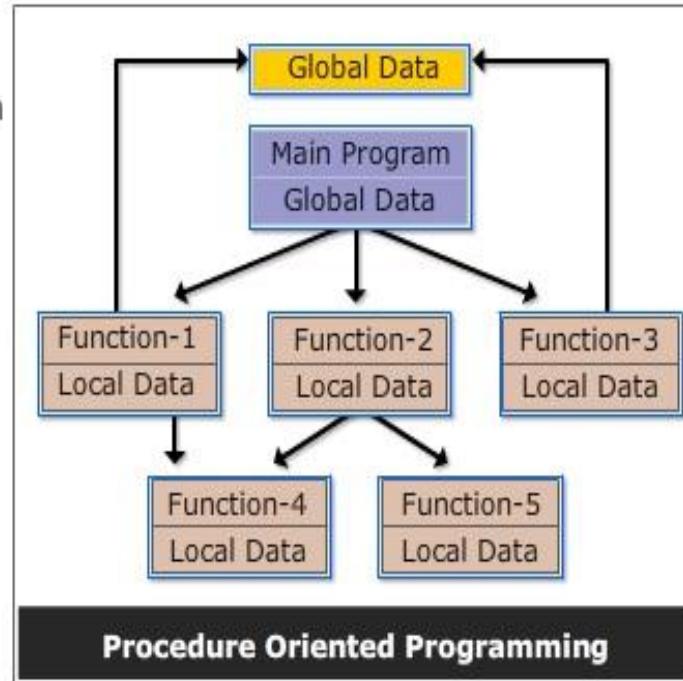
Introduction to POP

- The procedure oriented programming characterizes a program as a series of user-defined functions
- Given problem is split into more manageable modules called as functions
- All knowledge about the system is built into a set of functions

Limitations of POP:

- Focus is on procedures rather than data
- Employs top-down approach in program design
- Large procedural-based programs to turn into spaghetti-code
- Function and data structure do not model the real world
- It is difficult to create new data types
- Global variables have no access restriction
- Solution is solution-domain specific

Examples: BASIC, COBOL, C, FORTRAN

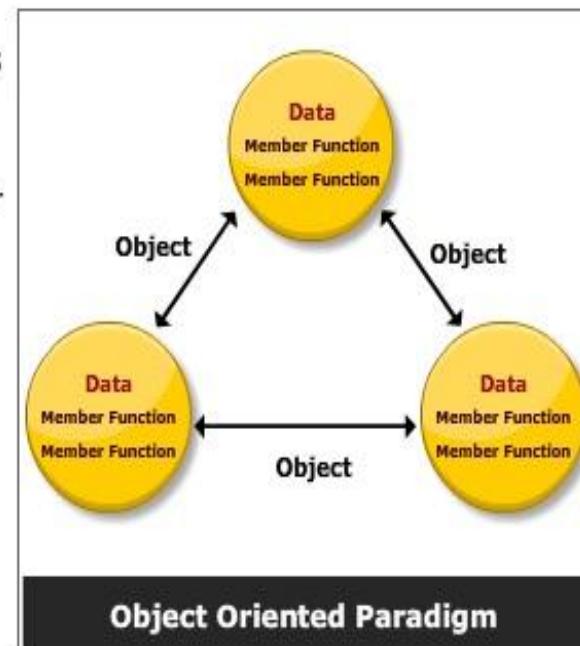


Introduction to Object Oriented Programming

- OOP is a programming paradigm using **objects** consisting of **data fields** and **methods** together with their interactions
- OOP represents an attempt to make programs, more closely model the way people think and deal with the real world objects
- Object-oriented technology is built upon object models
- Object model is defined by classes and objects
- An object is a real world entity with state and behavior

Object Oriented Concepts:

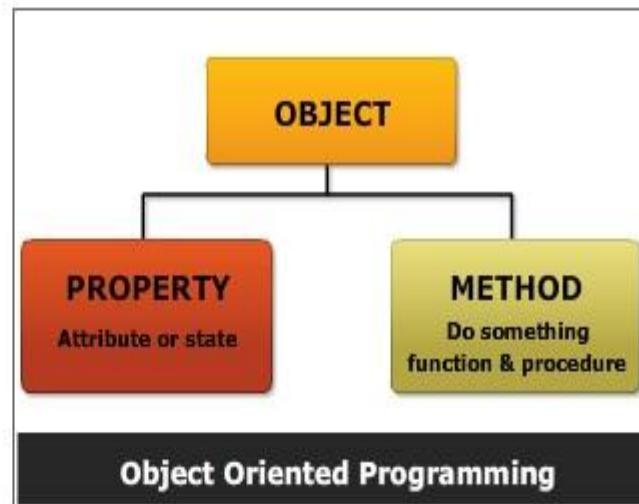
- Objects and classes
- Inheritance
- Encapsulation
- Polymorphism and dynamic binding



Advantages of OOP

- Focus is on data rather than on functions or procedures
- Follows bottom-up approach in program design
- Program is organized by using classes and objects
- Complexity is reduced by modeling software objects to real world objects
- Solution is problem-domain specific
- Makes it easy to maintain and modify existing code
- OOP provides a good framework for code libraries

Examples: Java, C++, C#, Smalltalk, Simula



Introduction to Objects

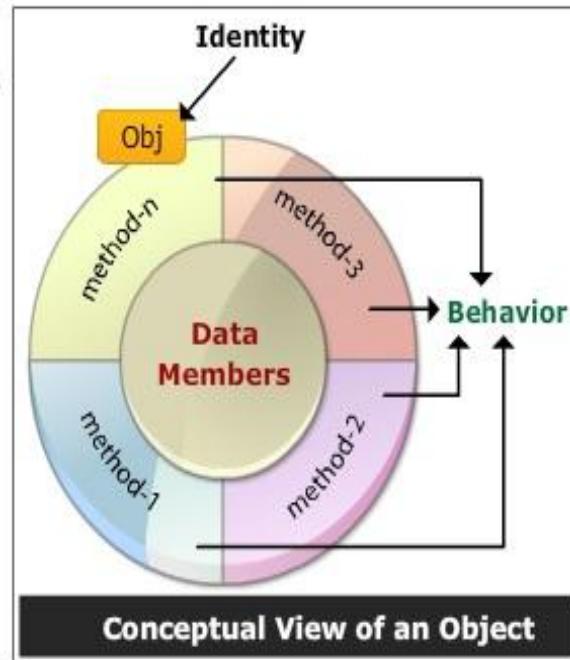
Definition:

- An object is a bundle of variables and related methods
- Everything in the world is an object
- Software objects are modeled after real-world objects

Properties of object:

- **Identity** of the object
- Object stores its **state** in fields or variables
- Exposes its **behavior** through methods or functions

An object is also known as instance which refers to a particular object



Example

Example: Daniel's car

Properties:

- **Identity:** Registration number (ex: XS29BB0106)
- **State:** Color, Make, Model, Mileage, Speed
- **Behavior:** Start, Stop, Brake, Change Speed
- The variables of an object are formally known as instance variables
- Instance variables contain the state for a particular object or instance
- State of Daniel's car
 - **Color:** Ash
 - **Make:** BMW
 - **Speed:** 50kmph



Example of an Object

Introduction to Classes

Definition: A class is a blueprint that defines the variables and the methods common to all objects of a certain kind

- A class is collection of objects of similar types
- A class is used to manufacture or create objects
- The class declares the instance variables necessary to contain the state of every object
- Any number of objects can be created from a class
- Each object of a given class contains the structure and behavior defined by the class
- A class is an *abstract data type* which declares *variables and methods* to access the variable

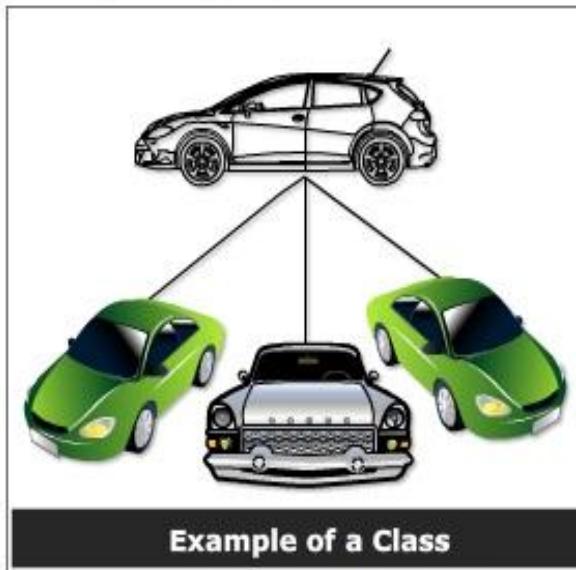
Example

Introduction to Classes

X

Example: A Car class

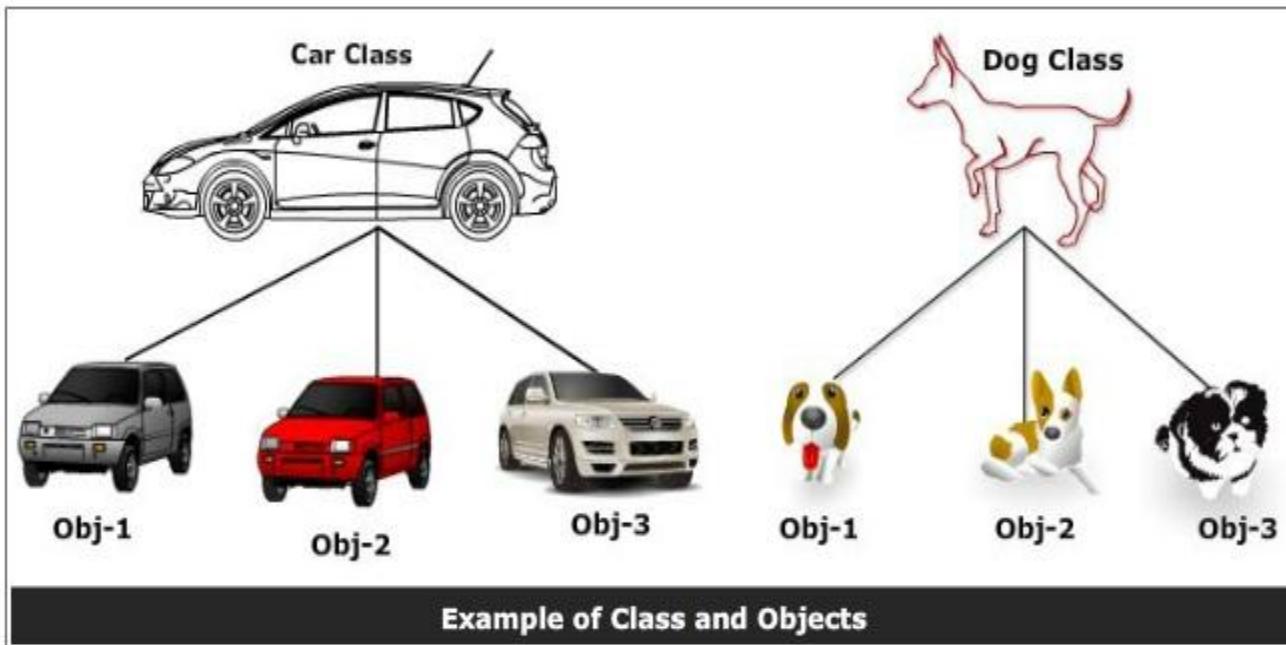
- **Variables:** Color, Make, Model, Mileage, Speed
- **Methods:** Start, Stop, Brake, Change Speed



Example

Classes and Objects

- A class is a blueprint that defines the variables and the methods, common to all objects of a certain kind
- An object is a bundle of variables and related methods defined by the class



Introduction to Abstraction

Abstraction is the process of hiding the irrelevant details and exposing only the essential features of a particular object

- Abstraction is achieved with objects
- Abstraction focuses on **what the object does** instead of how it does
- Complexity is managed through abstraction

Example: Daniel's car

Irrelevant details to Daniel: Engine functionality, Accelerator functionality

Relevant details to Daniel: Make, Model, Gears, Brakes, Accelerator, Steering



Complete View of Car Object

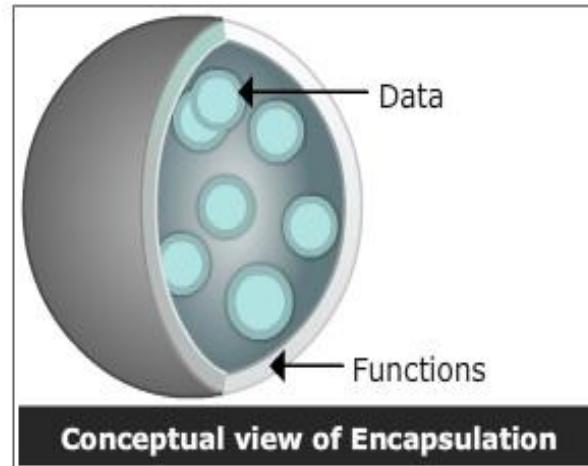


Daniels View of Car Object

Introduction to Encapsulation

Wrapping up of data and function into a single entity is called encapsulation

- Encapsulation is achieved by keeping data and the behavior in one capsule that is class
- Access to the code and data inside the wrapper is tightly controlled through a well-defined interface
- Encapsulation provides a layer of security around manipulated data, protecting it from external interference and misuse

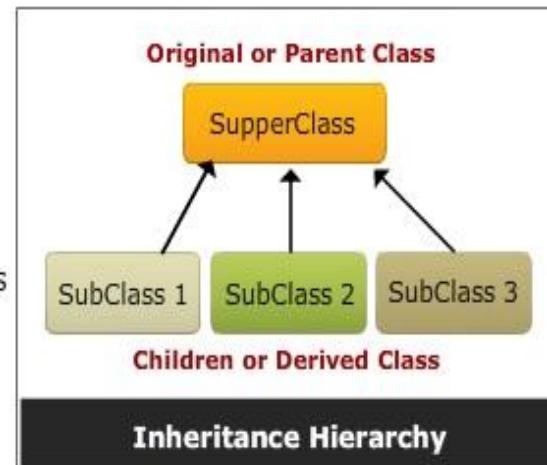


Advantages:

- Restricts access to data
- Implementation independence

Introduction to Inheritance

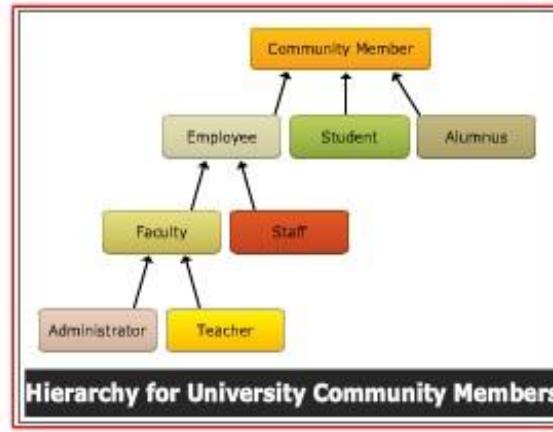
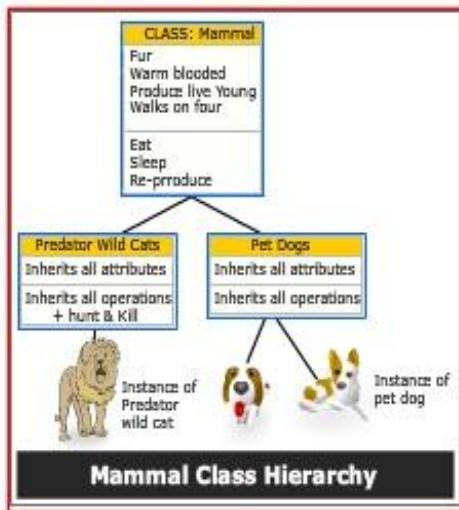
- Inheritance is the process of acquiring the properties from one class object to another class object
- Minimum two classes will be involved in inheritance
 - 1) Super Class
 - 2) Sub Class
- Sub class object will have added features of super class
- A sub class inherits all the attributes and behaviors of the super class and may have additional attributes and behaviors



Example

X

- Inheritance is a way to reuse the code of existing objects or establishing a subtype from an existing object or both
- Inheritance leads to hierachial classification, which is also called as "**is a**" relationship



Types of Inheritance

There are five different inheritances supported in C++:

Single Inheritance

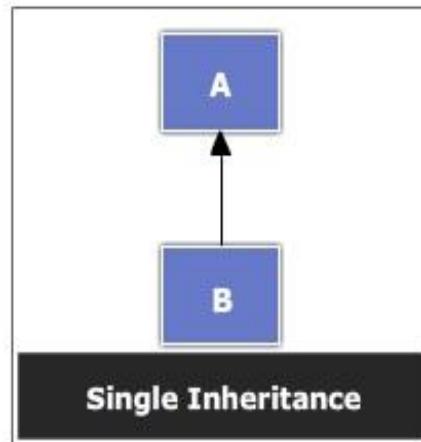
Multilevel Inheritance

Hierarchical Inheritan

Multiple Inheritance

Hybrid Inheritance

- A derived class with only one base class is called Single Inheritance



Note: The direction of arrow indicates the direction of inheritance

Types of Inheritance

There are five different inheritances supported in C++:

Single Inheritance

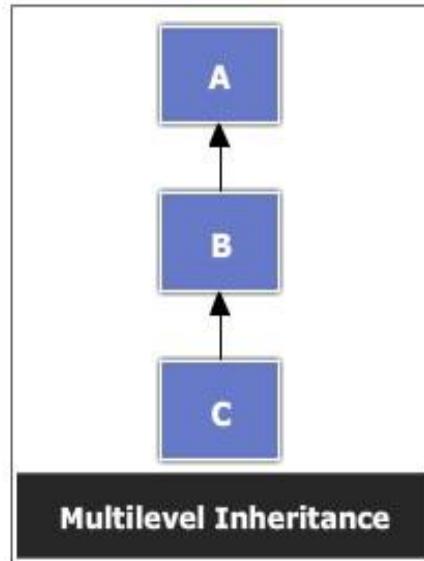
Multilevel Inheritance

Hierarchical Inheritance

Multiple Inheritance

Hybrid Inheritance

- The process of deriving a class from another derived class is called Multilevel Inheritance



Note: The direction of arrow indicates the direction of inheritance

Types of Inheritance

There are five different inheritances supported in C++:

Single Inheritance

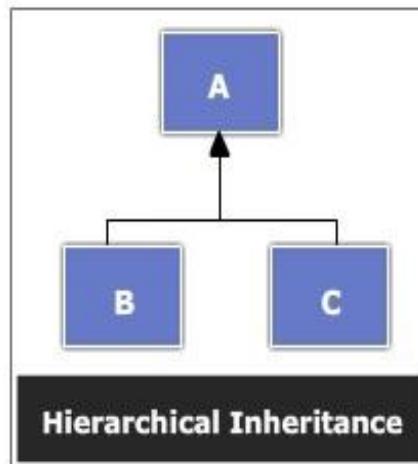
Multilevel Inheritance

Hierarchical Inheritance

Multiple Inheritance

Hybrid Inheritance

- The properties of one class are inherited by more than one class is called Hierarchical



Note: The direction of arrow indicates the direction of inheritance

Types of Inheritance

There are five different inheritances supported in C++:

Single Inheritance

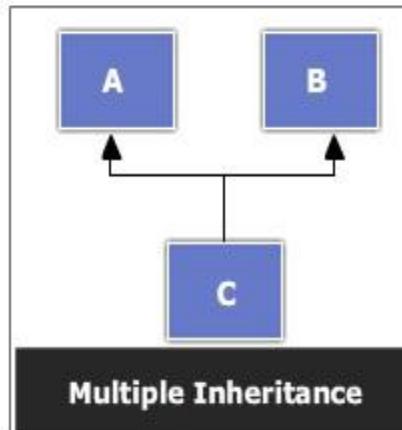
Multilevel Inheritance

Hierarchical Inheritance

Multiple Inheritance

Hybrid Inheritance

- A derived class with more than one base class is called Multiple Inheritance



Note: The direction of arrow indicates the direction of inheritance

Types of Inheritance

There are five different inheritances supported in C++:

Single Inheritance

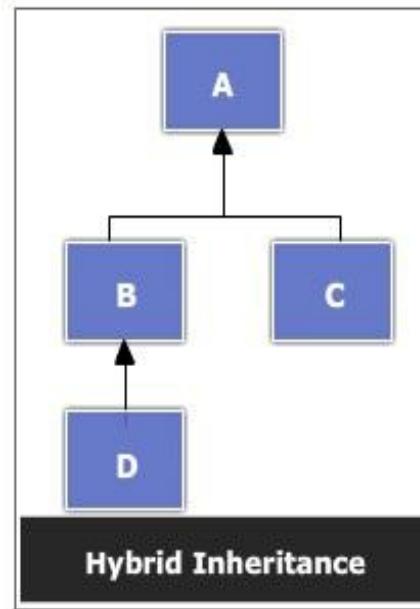
Multilevel Inheritance

Hierarchical Inheritance

Multiple Inheritance

Hybrid Inheritance

- The combination of two or more types of inheritance is called Hybrid Inheritance



Note: The direction of arrow indicates the direction of inheritance

Polymorphism



by Sinipull for codecall.net

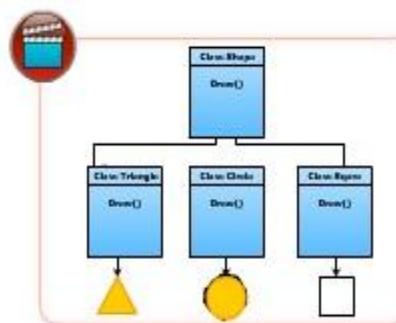
Introduction to Polymorphism



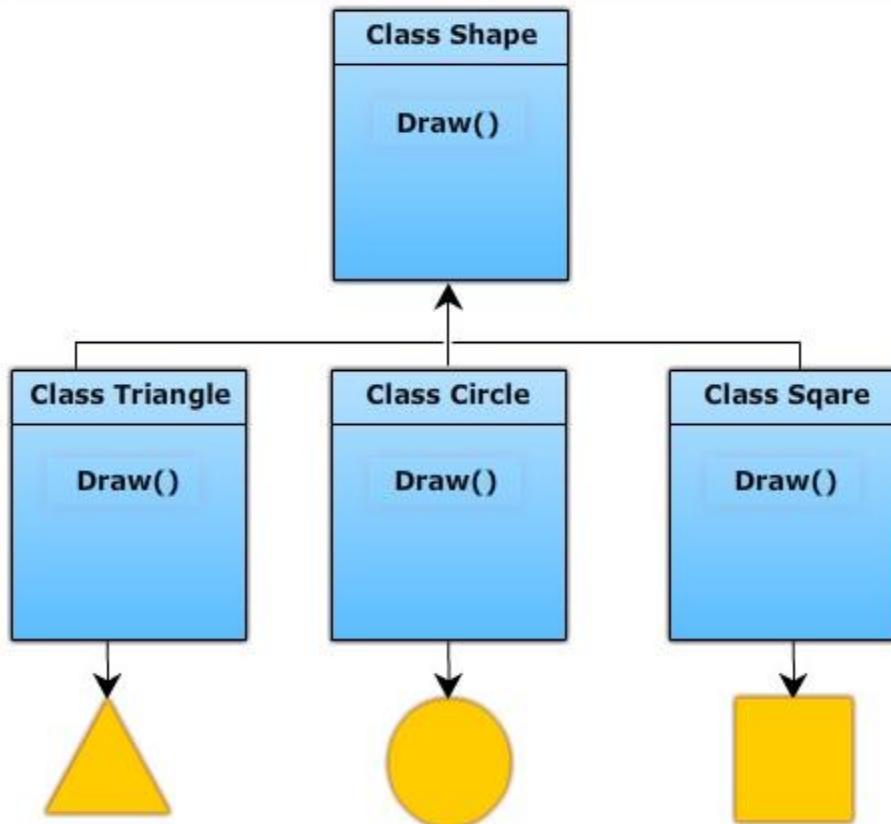
- The ability to take more than one form is called polymorphism
- It allows different objects to respond to the same message in different ways and the response is specific to the object
- An operation may exhibit different behaviors in different instances
- The behavior depends upon the types of data used in the operation

Example: Consider addition (+) operation

- The behavior depends upon the types of data used in the operation
 - For two numbers, it generates a sum
 - For two strings, it generates a third string
- It has same operation but different behaviors



|| Polymorphism



00:16 / 00:16 1x

Dynamic Binding

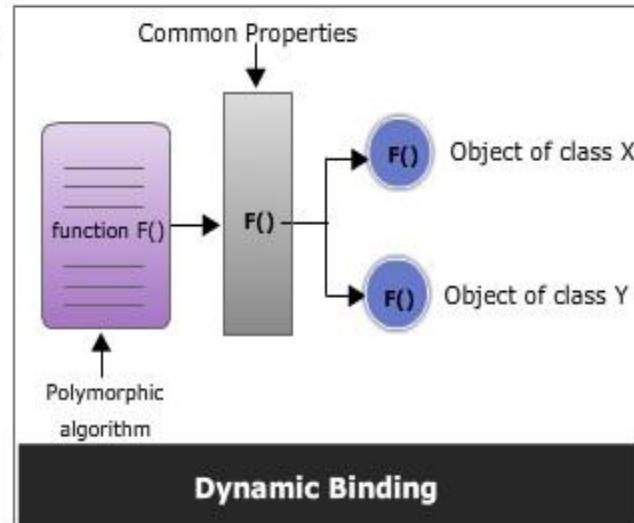
- Binding refers to linking of a procedure call to the code which is to be executed, in response to the call

Static Binding or Early Binding:

- When the function call is resolved at compile time, it is called as static binding
- Static binding applies to all 'C' function calls

Dynamic Binding or Late Binding :

- Refers to the case where compiler is not able to resolve the call and the binding is done at run time only
- If the function call is resolved at run time then it is called as dynamic binding
- It applies when the function pointers are called
- It is associated with polymorphism and inheritance



What is an Object?

- Real world entities or things which have:
 - 1) State
 - 2) Behavior
 - 3) Identity

Example: your dog, your car etc.,

- State – name, color, breed of a dog
 - Behavior – sitting, barking, wagging tail, running
 - Identity – your dog
-
- A software object is a bundle of variables (state) and methods (operations).

What is a Class?

- Class is basis for the Java language.
- Each concept we wish to describe in Java must be included in a class.
- A class is a template for objects
- A class is a blueprint / prototype that defines the variables and methods common to all objects of a certain kind.
- **Example:** 'your dog' is a object of the class Dog.
- An object holds values for the variables defined in the class.
- A class defines a new data type, whose values are objects
- An object is an instance of a class

The Class hierarchy

- In Java , classes are arranged in a hierarchy
- The root, or topmost class is **Object**
- Every class but Object has at least one super class
- A class may have subclasses
- Each class *inherits* all the fields and methods of its super classes

Class Definition

- **A class consists of :**
 - Name,
 - Several variable declarations (class-instance variables)
 - Several method declarations
- A Class also consists constructors and blocks
- **General form of a class:**

```
class ClassName {  
    type instance-variable-1;  
    ...  
    type instance-variable-n;  
  
    type method-name-1(parameter-list) { ... }  
    type method-name-2(parameter-list) { ... }  
    ...  
    type method-name-m(parameter-list) { ... }  
}
```

Class, Instance and Local Variables

	Local variable	Instance variable	Class / static variable
Declaration	Inside a method/constructor or as formal parameters	Inside a class, but outside methods/constructors	Inside a class with 'static', but outside methods/constructors
Usage and Lifetime	Used by methods to hold intermediate results. Created when the method is entered and destroyed on exit.	A separate copy is created for each object using the keyword new and destroyed when no more reference is made by any object.	One single copy of variable shared by all objects is created when the program starts and destroyed when the program ends.
Example: Class of Car2	<pre>Public void moveForward(int dist) { double gasUsed = Math.abs(dist) / 100.0 * gasMileage;</pre>	<pre>String owner = "NoName"; ColorImage carImage = new ColorImage("Car1.png"); double gasMileage = 10.0;</pre>	static final int NUMOFWHEEL = 4;
Initialization	Must be initialized before use.	Initialized to default values. E.g. 0 for numbers and null for objects.	Initialized to default values. E.g. 0 for numbers and null for objects.
Scope and Visibility	Visible only in the method/constructor or block in which they are declared. Access modifiers cannot be used.	Can be seen by all methods /constructors in the class. Visibility to other classes depends on access modifiers. They are often declared a private to protect them from being accidentally changed.	Same as instance variables but often declared as constants.

Declaring and creating objects

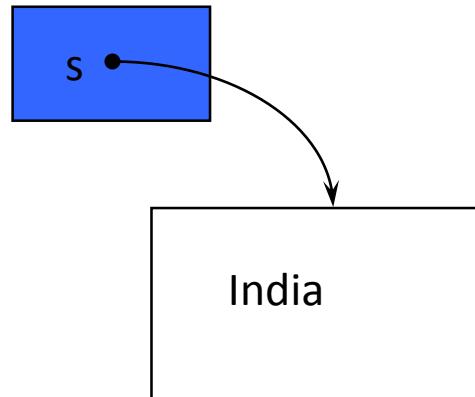
- **Declare a reference**

Example: Person p;
String s;

- **Creating an instance/object**

Person p = **new** Person();
String s = **new** String ("India");

- The **new** keyword is used to allocate memory at run-time

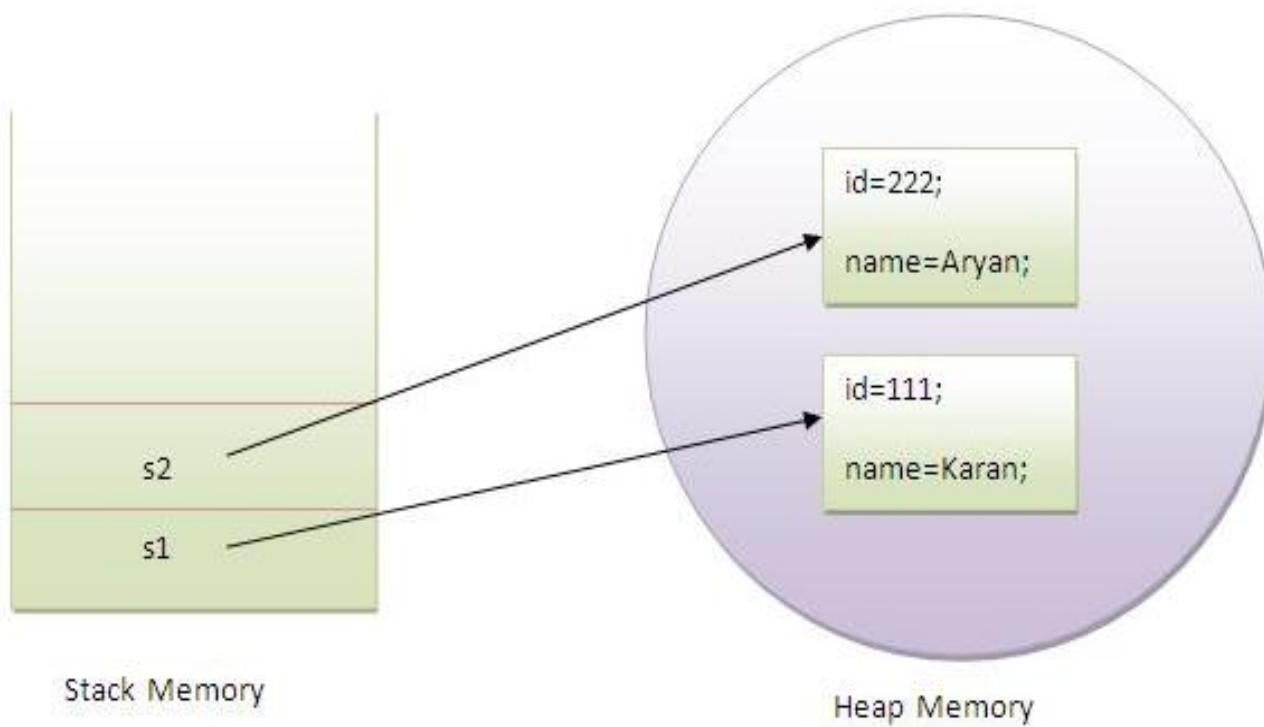


Example Program

```
class Student{  
    int rollno;  
    String name;  
  
    void insertRecord(int r, String n){ //method  
        rollno=r;  
        name=n;  
    }  
  
    void displayInformation(){System.out.println(rollno+" "+name);}  
  
    public static void main(String args[]){  
        Student s1=new Student();  
        Student s2=new Student();  
  
        s1.insertRecord(111,"Karan");  
        s2.insertRecord(222,"Aryan");  
  
        s1.displayInformation();  
        s2.displayInformation();  
    }  
}
```

[Student.java](#)

What happens in the memory?



Anonymous object

Anonymous object

Anonymous simply means nameless. An object that have no reference is known as anonymous object. If you have to use an object only once, anonymous object is a good approach.

```
class Calculation{  
  
    void fact(int n){  
        int fact=1;  
        for(int i=1;i<=n;i++){  
            fact=fact*i;  
        }  
        System.out.println("factorial is "+fact);  
    }  
  
    public static void main(String args[]){  
        new Calculation().fact(5);//calling method with anonymous object  
    }  
}
```

Output:Factorial is 120

Object Destruction

A program accumulates memory through its execution.

Two mechanisms to free memory that is no longer needed by the program:

- 1) Manual – in C/C++
- 2) Automatic – in Java

In Java, when an object is no longer accessible through any variable, it is eventually removed from the memory by the garbage collector.

Garbage collector is parts of the Java Run-Time Environment.

Constructor

A constructor used to initialize the state of an object.

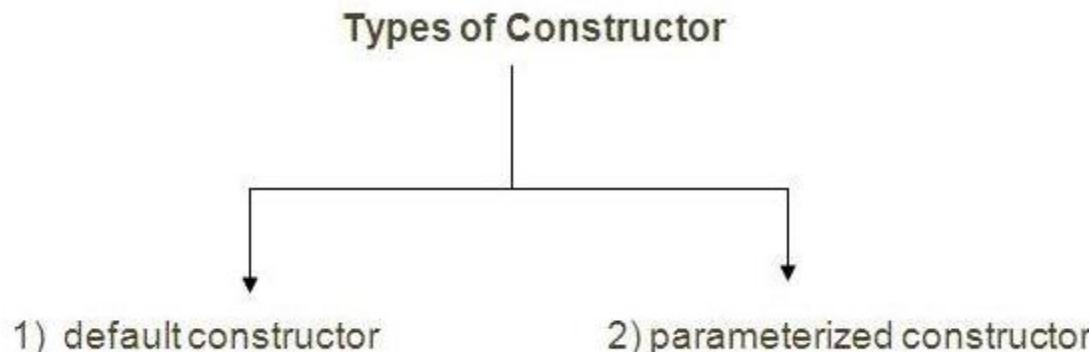
It is invoked at the time of object creation

It constructs the values (data) for the object

Features:

- 1) It is syntactically similar to a method
- 2) It has the same name as the name of its class
- 3) It is written without return type;
The default return type is the class

When the class has no constructor, the default constructor automatically supplied the compiler.



Default Constructor

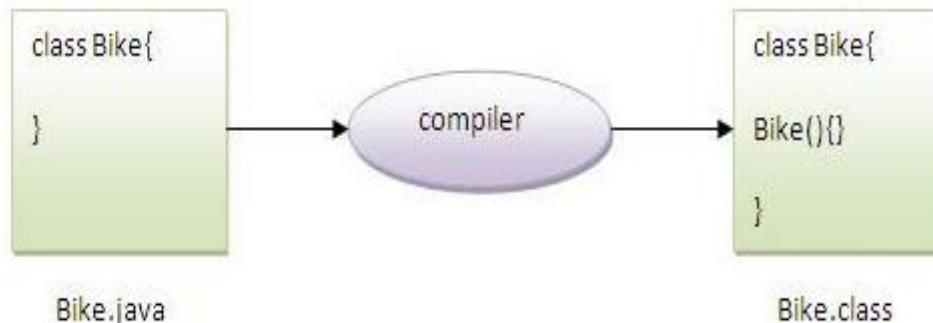
```
//<b><i>example of default constructor</i></b>
class Bike{

Bike(){System.out.println("Bike is created");}

public static void main(String args[]){
Bike b=new Bike();
}
}
```

Output: Bike is created

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Parameterized Constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
class Student{  
    int id;  
    String name;  
  
    Student(int i, String n){  
        id = i;  
        name = n;  
    }  
    void display(){System.out.println(id+" "+name);}  
  
    public static void main(String args[]){  
        Student s1 = new Student(111, "Karan");  
        Student s2 = new Student(222, "Aryan");  
        s1.display();  
        s2.display();  
    }  
}
```

Output: 111 Karan
222 Aryan

Constructor Overloading

```
//<b><i>Program of constructor overloading</i></b>

class Student{
    int id;
    String name;
    int age;
    Student(int i,String n){
        id = i;
        name = n;
    }
    Student(int i,String n,int a){
        id = i;
        name = n;
        age=a;
    }
    void display(){System.out.println(id+" "+name+" "+age);}

    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan",25);
        s1.display();
        s2.display();
    }
}
```

Constructor – Copying values

```
//<b><i>Program of Copying the values of one object to another</i></b>

class Student{
    int id;
    String name;
    Student(int i, String n){
        id = i;
        name = n;
    }

    Student(Student s){
        id = s.id;
        name = s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(s1);
        s1.display();
        s2.display();
    }
}
```

Constructors vs. Methods

There are many differences between constructors and methods. They are given below.

Constructor	Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Method Overloading

- A class with multiple methods by the **same name** but **different parameters**
- Implements polymorphism in java (static)
- **Three ways:**
 - Number of arguments
 - Types of arguments
 - Order and Type of argument

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

```
class Calculation{  
    void sum(int a,int b){System.out.println(a+b);}  
    void sum(int a,int b,int c){System.out.println(a+b+c);}  
  
    public static void main(String args[]){  
        Calculation obj=new Calculation();  
        obj.sum(10,10,10);  
        obj.sum(20,20);  
  
    }  
}
```

Output:30

40

Method Overloading

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two double arguments.

```
class Calculation{
    void sum(int a,int b){System.out.println(a+b);}
    void sum(double a,double b){System.out.println(a+b);}

    public static void main(String args[]){
        Calculation obj=new Calculation();
        obj.sum(10.5,10.5);
        obj.sum(20,20);

    }
}
```

Output:21.0

Method overloading is not possible by changing return types.

Que) Why Method Overloading is not possible by changing the return type of method?

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur:

because there was problem:

```
class Calculation{  
    int sum(int a,int b){System.out.println(a+b);}  
    double sum(int a,int b){System.out.println(a+b);}  
  
    public static void main(String args[]){  
        Calculation obj=new Calculation();  
        int result=obj.sum(20,20); //Compile Time Error  
    }  
}
```

int result=obj.sum(20,20); //Here how can java determine which sum() method should be called

Is Overloading the main() method possible ?

Overloading main() method

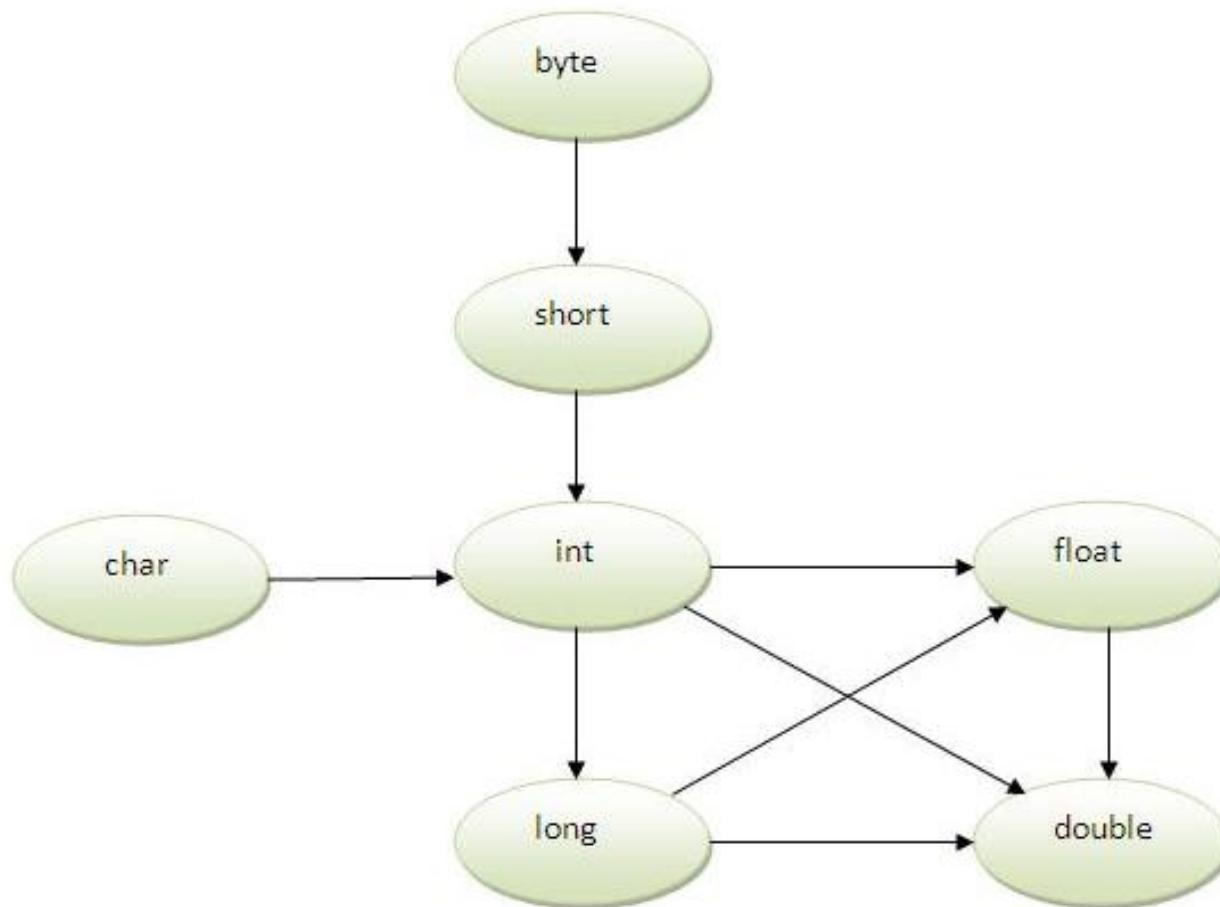
```
public class Simple /*Whenever your class is public and contains main()
method, file name must be same as your class name.*/
{
    public static void main(String[] args)
    {
        System.out.println("Hello World!");
        main(10); // main() call
    }
    public static void main(int a) // main() method overloading
    {
        System.out.println(a);
    }
}
```

Simple.java

Method Overloading

Method Overloading and TypePromotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



Method Overloading

Example of Method Overloading with TypePromotion

```
class Calculation{  
    void sum(int a,long b){System.out.println(a+b);}  
    void sum(int a,int b,int c){System.out.println(a+b+c);}  
  
    public static void main(String args[]){  
        Calculation obj=new Calculation();  
        obj.sum(20,20); //now second int literal will be promoted to long  
        obj.sum(20,20,20);  
    }  
}
```

Output : 40

60

Parameter Passing

Only **pass-by value or call by value** is available in Java. There is no **call by reference**. Primitive data types and objects can be passed as **values**

Example of call by value in java

In case of call by value original value is not changed. Let's take a simple example:

```
class Operation{  
    int data=50;  
  
    void change(int data){  
        data=data+100;//changes will be in the local variable only  
    }  
  
    public static void main(String args[]){  
        Operation op=new Operation();  
  
        System.out.println("before change "+op.data);  
        op.change(500);  
        System.out.println("after change "+op.data);  
    }  
}
```

Operation.java

Parameter Passing

Another Example of call by value in java

In case of call by reference original value is changed if we made changes in the called method. If we pass object in place of any primitive value, original value will be changed. In this example we are passing object as a value. Let's take a simple example:

```
class Operation2{  
    int data=50;  
  
    void change(Operation op){  
        op.data=op.data+100;//changes will be in the instance variable  
    }  
  
    public static void main(String args[]){  
        Operation2 op=new Operation2();  
  
        System.out.println("before change "+op.data);  
        op.change(op);//passing object  
        System.out.println("after change "+op.data);  
    }  
}
```

Operation1.java

Session 3:

Learning Objectives

- **Explain this facility**
- **Describe static member, method and block**
- **JDK and its usage**
- **Garbage Collection**

this keyword

- **this** is a reference variable that refers to the current object
- Call to **this()** must be always **first**

Usage:

1. Used to refer current class instance variable
2. **this()** is used to invoke current class constructor
3. Used to invoke current class method
4. Can be passed as an argument to the method
5. Used as argument in constructor call
6. Used to return current class instance

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
class student{
    int id;
    String name;

    student(int id, String name){
        id = id;
        name = name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        student s1 = new student(111, "Karan");
        student s2 = new student(321, "Aryan");
        s1.display();
        s2.display();
    }
}
```

using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

<i>example of **this** keyword</i>

```
class Student{
    int id;
    String name;

    student(int id,String name){
        this.id = id;
        this.name = name;
    }
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        s1.display();
        s2.display();
    }
}
```

[ThisStudent.java](#)

this() is used to invoke current class constructor

```
class Student{
    int id;
    String name;
    Student (){System.out.println("default constructor is invoked");}
    Student(int id, String name){
        this(); //it is used to invoked current class constructor.
        this.id = id;
        this.name = name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        Student e1 = new Student(111,"karan");
        Student e2 = new Student(222,"Aryan");
        e1.display();
        e2.display();
    }
}
```

Output:

```
default constructor is invoked
default constructor is invoked
```

Advantage of this()

```
class Student{  
    int id;  
    String name;  
    String city;  
  
    Student(int id, String name){  
        this.id = id;  
        this.name = name;  
    }  
    Student(int id, String name, String city){  
        this(id, name); //now no need to initialize id and name  
        this.city=city;  
    }  
    void display(){System.out.println(id+" "+name+" "+city);}  
  
    public static void main(String args[]){  
        Student e1 = new Student(111,"karan");  
        Student e2 = new Student(222,"Aryan","delhi");  
        e1.display();  
        e2.display();  
    }  
}
```



Rule: Call to this() must be the first statement in constructor.

```
class Student{
    int id;
    String name;
    Student (){System.out.println("default constructor is invoked");}

    Student(int id, String name){
        id = id;
        name = name;
        this(); //must be the first statement
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
        Student e1 = new Student(111, "karan");
        Student e2 = new Student(222, "Aryan");
        e1.display();
        e2.display();
    }
}
```

Output: Compile Time Error

3)The this keyword can be used to invoke current class method (implicitly).

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example

[copy to clipboard](#)

```
class S{  
    void m(){  
        System.out.println("method is invoked");  
    }  
    void n(){  
        this.m(); //no need because compiler does it for you.  
    }  
    void p(){  
        n(); //compiler will add this to invoke n() method as this.n()  
    }  
    public static void main(String args[]){  
        S s1 = new S();  
        s1.p();  
    }  
}
```

4) The this keyword can be passed as an argument in the method.

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling.
Let's see the example:

```
copy to clipboard
class S{
    void m(S obj){
        System.out.println("method is invoked");
    }
    void p(){
        m(this);
    }

    public static void main(String args[]){
        S s1 = new S();
        s1.p();
    }
}
```

Output:method is invoked

6) The this keyword can be used to return current class instance.

We can return the this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
return_type method_name(){  
    return this;  
}
```

Example of this keyword that you return as a statement from the method

[copy to clipboard](#)

```
class A{  
A getA(){  
return this;  
}  
void msg(){System.out.println("Hello java");}  
}
```

```
class Test{  
public static void main(String args[]){  
new A().getA().msg();  
}
```

[**TestThis.java**](#)

Proving “this”

```
class A{  
void m(){  
System.out.println(this); //prints same reference ID  
}  
  
public static void main(String args[]){  
A obj=new A();  
System.out.println(obj); //prints the reference ID  
  
obj.m();  
}  
}
```

Output: A@13d9c02
A@13d9c02

The static keyword

- Java **methods** and **variables** can be declared **static**
- These will exist **independent of any object**

This means that a Class's

- **static methods** can be called even if **no objects** of that class have been created and
 - **static data** is “shared” by **all instances** (i.e., one value per class instead of one per instance)
-
- **Static**
 - means “global”—all objects refer to the same storage.
 - applies to variables or methods
 - **usage:**
 - with variable of a class
 - with a method of a class

Example of static variable

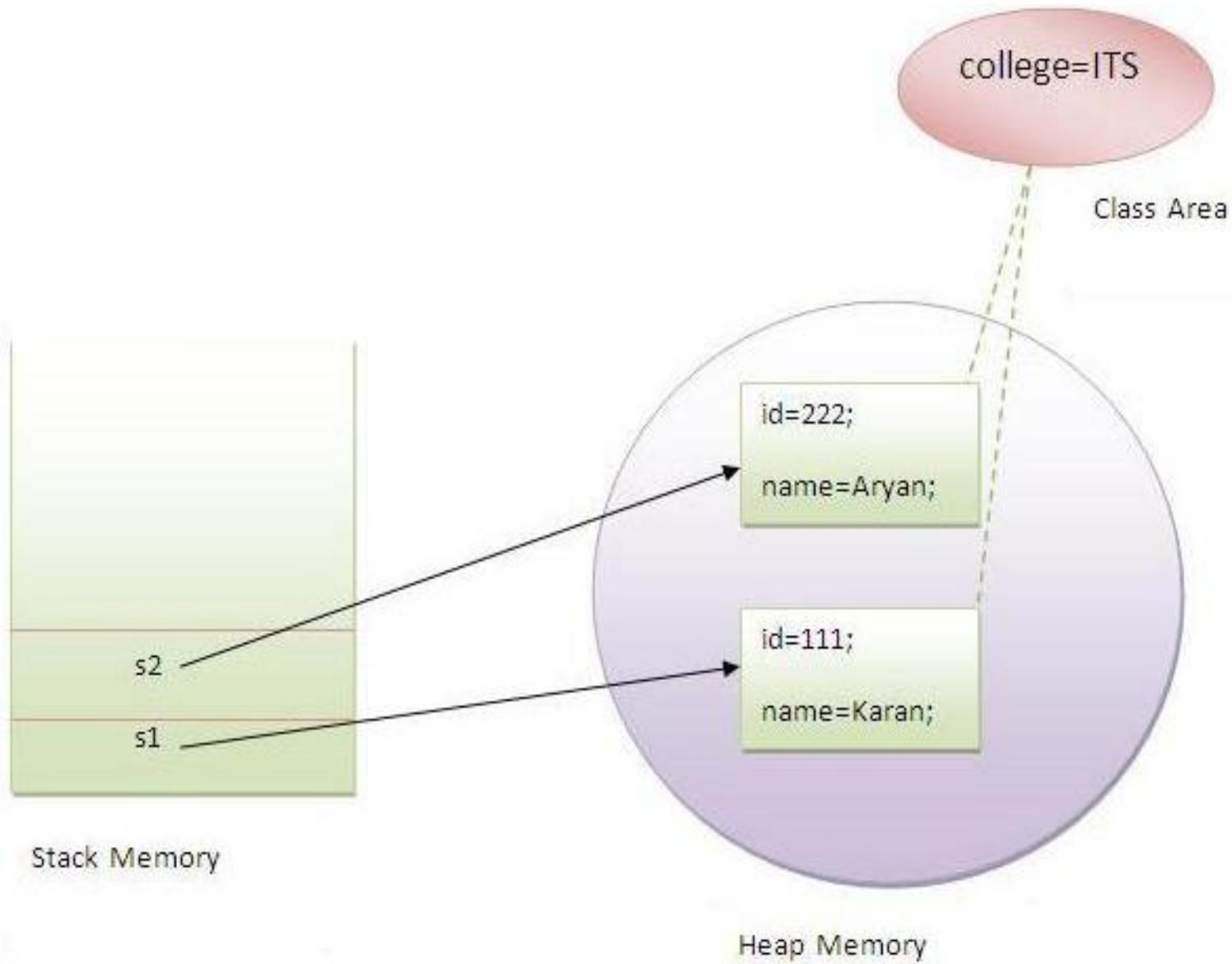
```
//<b>Program of static variable</b>

class Student{
    int rollno;
    String name;
    static String college = "ITS";

    Student(int r,String n){
        rollno = r;
        name = n;
    }
    void display (){System.out.println(rollno+" "+name+" "+college);}

    public static void main(String args[]){
        Student s1 = new Student (111,"Karan");
        Student s2 = new Student (222,"Aryan");

        s1.display();
        s2.display();
    }
}
```



Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
class Counter{  
    static int count=0;//will get memory only once and retain its value  
  
    Counter(){  
        count++;  
        System.out.println(count);  
    }  
  
    public static void main(String args[]){  
  
        Counter c1=new Counter();  
        Counter c2=new Counter();  
        Counter c3=new Counter();  
  
    }}  
 
```

Output : 1

2

3

Usage of Static Method

```
class Student{  
    int rollno;  
    String name;  
    static String college = "ITS";  
  
    static void change(){  
        college = "BBDIT";  
    }  
  
    Student(int r, String n){  
        rollno = r;  
        name = n;  
    }  
  
    void display (){System.out.println(rollno+" "+name+" "+college);}  
  
    public static void main(String args[]){  
        Student.change();  
  
        Student s1 = new Student (111,"Karan");  
        Student s2 = new Student (222,"Aryan");  
        Student s3 = new Student (333,"Sonoo");  
  
        s1.display();
```

Restrictions for static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

//Program of accessing non-static data member directly from static method main

```
class A{
    int a=40;//non static

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

Static Block

3)static block:

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```
//<b>Program of static block</b>

class A{

    static{System.out.println("static block is invoked");}

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

Output: static block is invoked
Hello main

<i>//Program of executing a program without main() method</i>

```
class A{
    static{
        System.out.println("static block is invoked");
        System.exit(0);
    }
}
```

```
public class SSS {
    static{
        System.out.println("Parent is:");
        System.exit(0);
    }
}
```

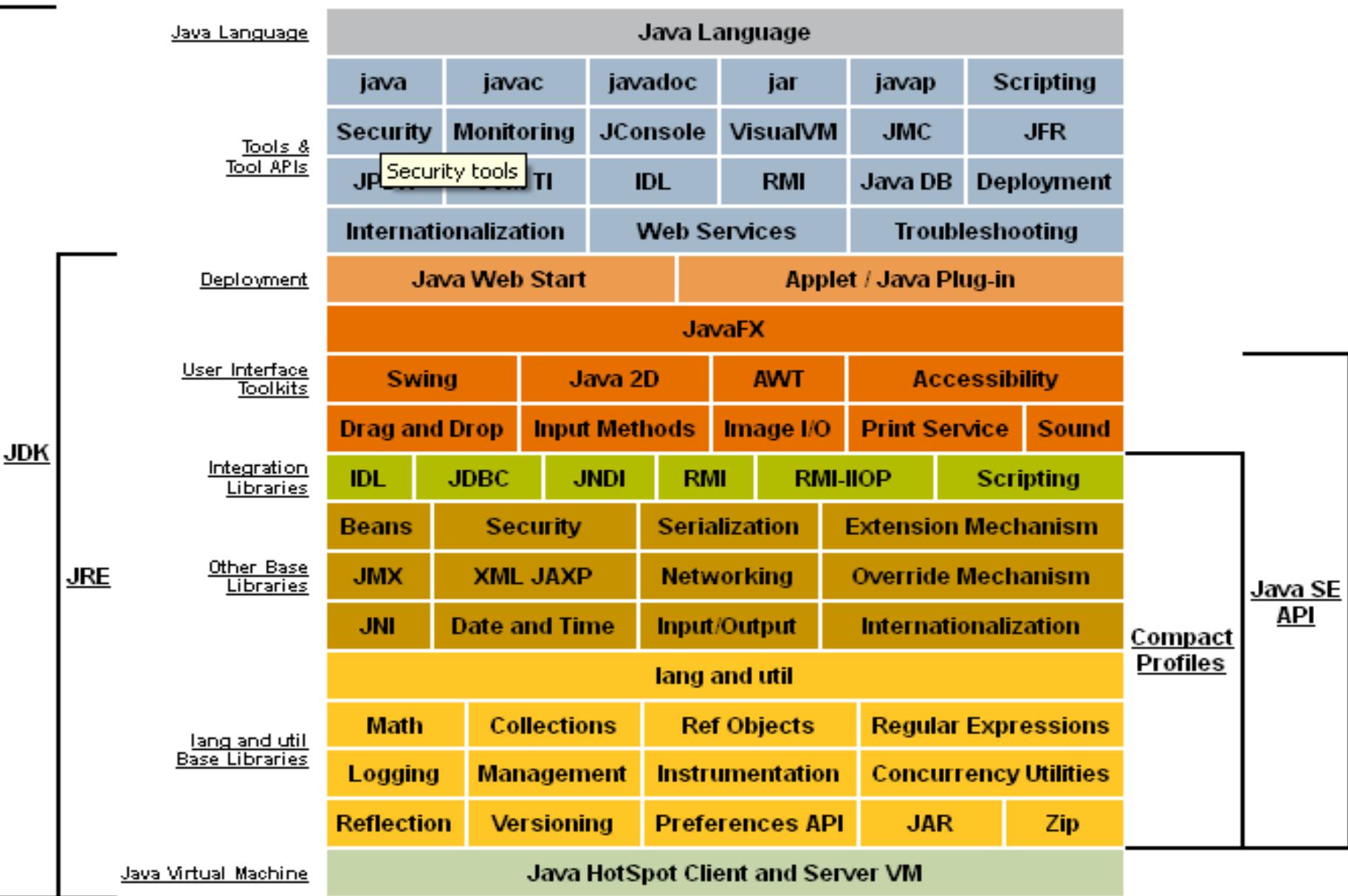
javac -deprecated SS.java

[SSS.java](#)

```
C:\Users\Greenivas Sadhu\Desktop>javac SS.java
C:\Users\Greenivas Sadhu\Desktop>java SS
Parent is:
Exception in thread "main" java.lang.NoSuchMethodError: main

C:\Users\Greenivas Sadhu\Desktop>javap SS
Compiled from "SS.java"
public class SS extends java.lang.Object{
    public SS();
    static {};
}
```

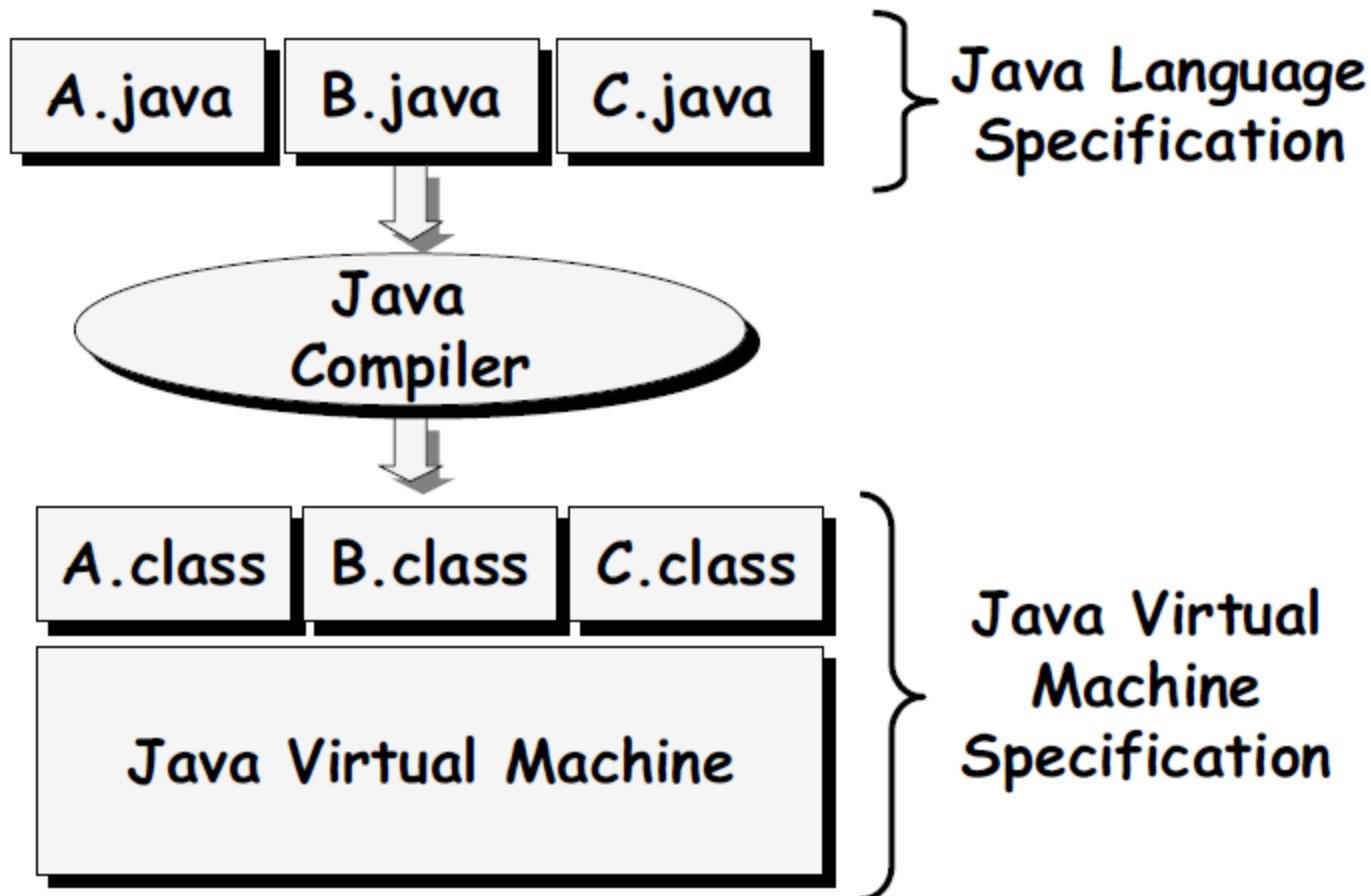
The Java Platform – JDK / Java SE



Java Development Kit

- javac - The Java Compiler
 - java - The Java Interpreter
 - appletviewer -Tool to run the applets
-
- javap - to print the Java byte codes
 - javadoc - documentation generator
 - javah - creates C header files

The Big Picture



JVM

Class File Example (1)

HelloWorld.java

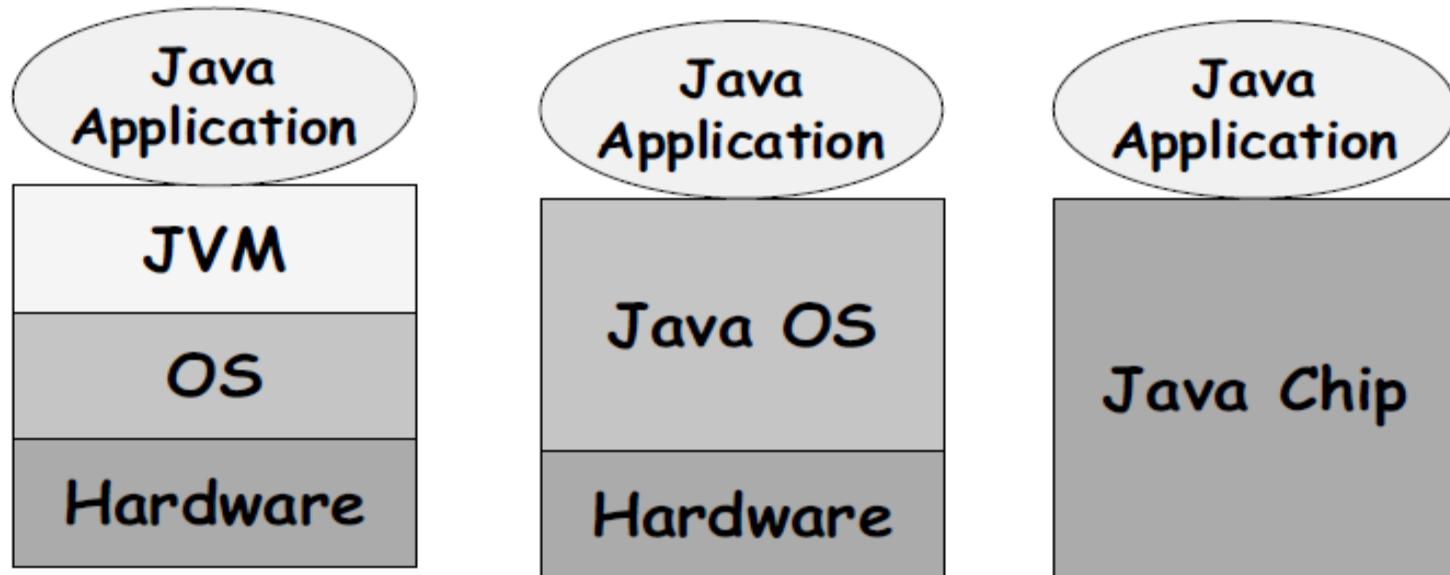
```
public class HelloWorld extends Object {  
    private String s;  
    public HelloWorld() {  
        s = 'Hello World!';  
    }  
    public void sayHello() {  
        System.out.println(s);  
    }  
    public static void main(String[] args) {  
        HelloWorld hello = new HelloWorld();  
        hello.sayHello();  
    }  
}
```

Class File Example (2)

HelloWorld.class

```
class HelloWorld  
Superclass java/lang/Object  
Constant Pool  
#0: 'Hello World'  
Fields  
    s descriptor : Ljava/lang/String;  
    modifiers : private  
Methods  
    <init> descriptor : ()V  
    modifiers : public  
    sayHello descriptor : ()V  
    modifiers : public  
    main descriptor : (Ljava/lang/String;)V  
    modifiers : public, static  
Bytecodes  
→ Bytecodes for <init> (the constructor)  
→ Bytecodes for sayHello  
→ Bytecodes for main
```

Implementations of JVM



JVM Specification does not specify
how a JVM is implemented

JVM

- **JVM** is a component of the Java system that interprets and executes the instructions in our class files.
- The following figure shows a block diagram of the JVM that includes its major **subsystems and memory areas**.

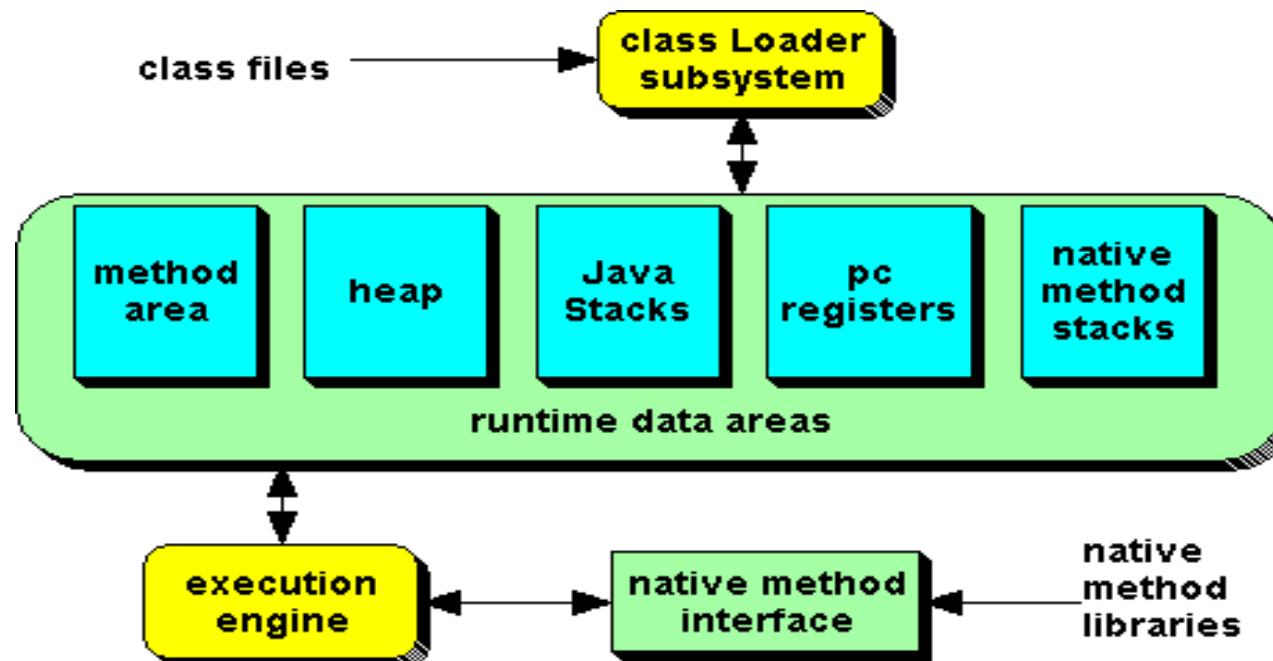
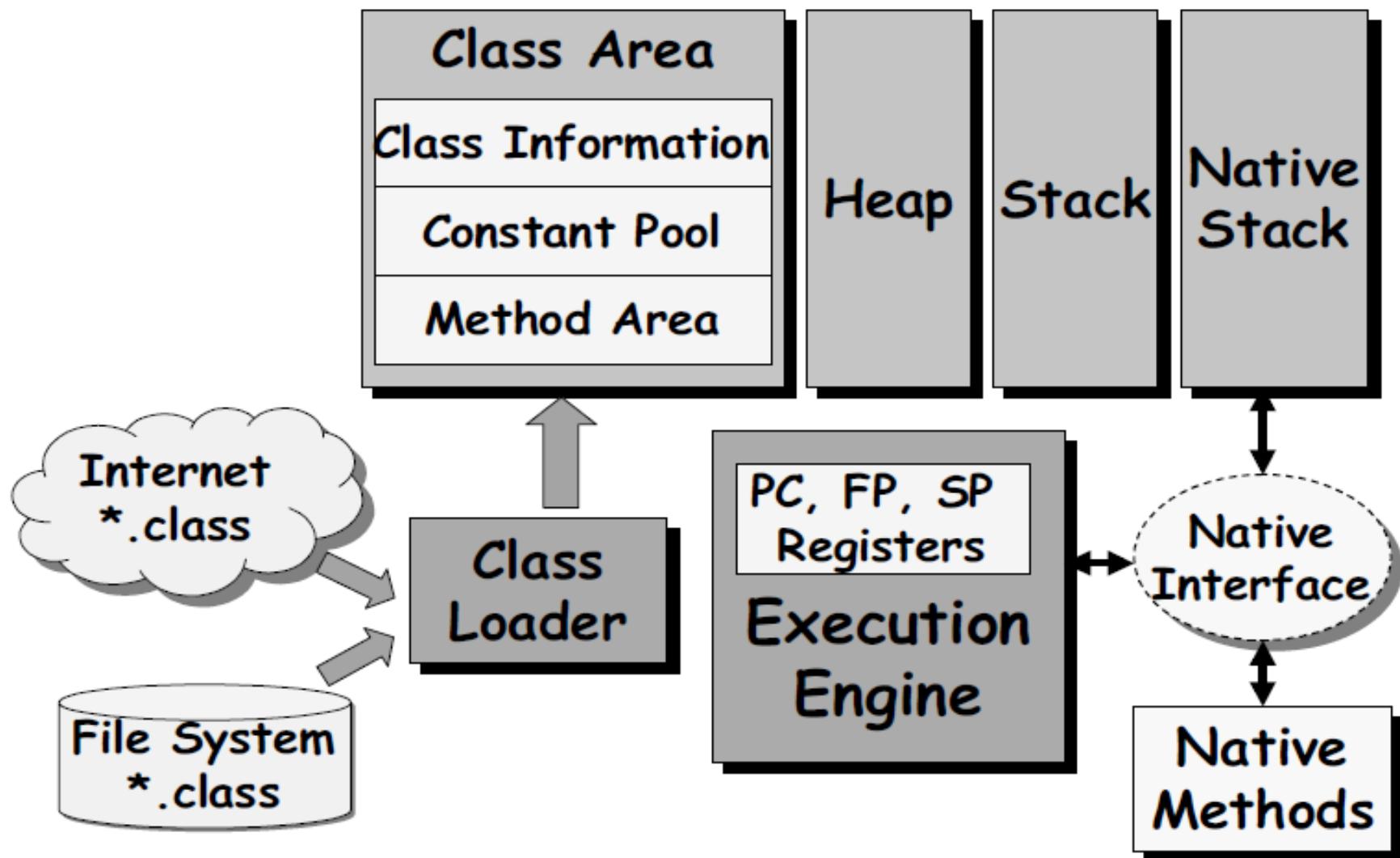


Figure 1: Memory configuration by the JVM.

JVM

- Each instance of the **JVM** has one **method area**, **one heap**, and **one or more stacks** - one for each thread
- When JVM loads a class file, it puts its information in the **method area**
- As the program runs, all **objects** instantiated are stored in the **heap**
- The **stack area** is used to **store activation records** as a program runs

Organization of JVM



Execution Engine

- Executes Java bytecodes either using interpreter or Just-In-Time compiler
- Registers:
 - PC: Program Counter
 - FP: Frame Pointer
 - SP: Operand Stack Top Pointer

Types of Class Loaders

Bootstrap Class Loader:

- Bootstrap class loader loads java's core classes like java.lang, java.util etc.
- These are classes that are part of java runtime environment.
- Bootstrap class loader is native implementation and so they may differ across different JVMs.

Extensions Class Loader:

- JAVA_HOME/jre/lib/ext contains jar packages that are extensions of standard core java classes.
- Extensions class loader loads classes from this ext folder
- **System Class Loader:**
Java classes that are available in the java classpath are loaded using System class loader.

The Class Loader Subsystem

- The class loader performs three main functions of JVM, namely: **loading, linking and initialization**
- The **linking process** consists of three sub-tasks, namely, **verification, preparation, and resolution**

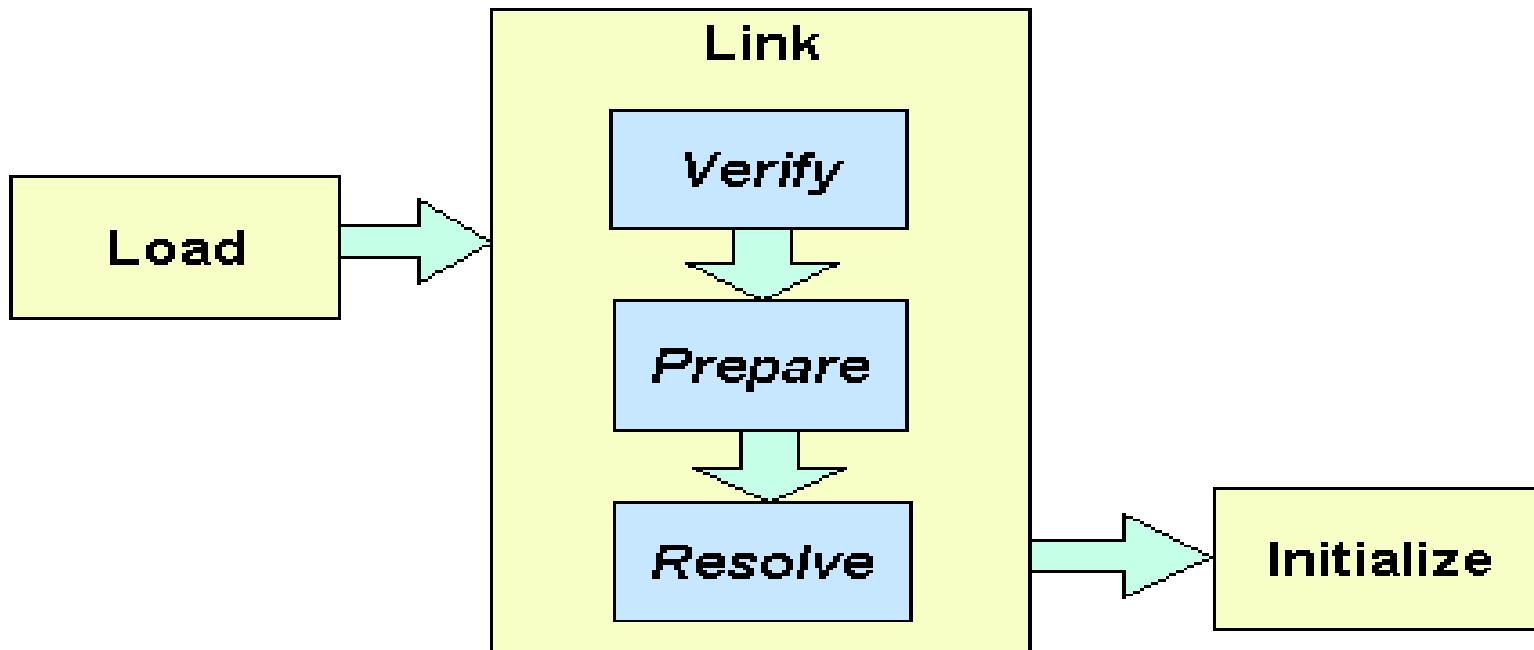


Figure 3: Class loading process.

Class Loader

1. Loading: finding and importing the binary data for a class
2. Linking:
 - Verification: ensuring the correctness of the imported type
 - Preparation: allocating memory for class variables and initializing the memory to default values
 - Resolution: transforming symbolic references from the type into direct references.
3. Initialization: invoking Java code that initializes class variables to their proper starting values

Class Loading Process

- **Loading** means reading the **class file for a type**, parsing it to get its information, and storing the information in the **method area**.
- For each type it loads, the **JVM** must store the following information in the method area:
 - The **fully qualified name** of the type
 - The fully **qualified name of the type's direct super class** or if the type is an **interface**, a list of its **direct super interfaces** .
 - Whether the type is a **class or an interface**
 - The type's **modifiers** (public, abstract, final, etc)
 - **Constant pool** for the type: constants and symbolic references.
 - **Field info** : name, type and modifiers of variables
 - **Method info**: name, return type, number & types of parameters, modifiers, byte codes, size of stack frame and exception table.

Class Loading Process (Cont'd)

- The end of the loading process is the creation of an instance of **java.lang.Class** for the **loaded type**.
- The purpose is to give access to some of the information captured in the method area for the type, to the programmer.
- Some of the methods of the class `java.lang.Class` are:

```
public String getName()
public Class getSupClass()
public boolean isInterface()
public Class[] getInterfaces()
public Method[] getMethods()
public Field[] getFields()
public Constructor[] getConstructors()
```

- Note that for any loaded type `T`, only one instance of `java.lang.Class` is created even if `T` is used several times in an application.
- To use the above methods, we need to first call the `getClass()` method on any instance of `T` to get the reference to the `Class` instance for `T`.

Verification During Linking Process

- The next process handled by the class loader is **Linking**.
- This involves three sub-processes:

Verification, Preparation and Resolution

- **Verification** is the process of ensuring **structurally correct** the **binary representation** of a class. The JVM has to make sure that a file it is asked to load was generated by **a valid compiler** and it is **well formed**
- Example of some of the things that are checked at verification are:
 - Every method is provided with a structurally **correct signature**
 - Every instruction obeys the **type discipline** of the Java language
 - Every branch instruction branches to the **start** not **middle** of another instruction

Preparation

- In this phase, the JVM allocates **memory** for the **class (i.e static) variables** and sets them to **default initial values**.
- Note that class variables are not initialized to their proper initial values until the initialization phase - **no java code is executed until initialization**.
- The default values for the various types are shown below:

Type	Initial Value
int	0
long	0L
short	(short) 0
char	'\u0000'
byte	(byte) 0
boolean	false
reference	null
float	0.0f
double	0.0d

Resolution

- Resolution is the process of **replacing symbolic** names for types, fields and methods used by a loaded type **with their actual references**.
- Symbolic references are resolved into a direct references by searching through the method area to locate the referenced entity.
- For the class below, at the loading phase, the class loader would have loaded the classes: TestClassClass, String, System and Object.

```
public class TestClassClass{  
    public static void main(String[] args){  
        String name = new String("ABC");  
        Class nameClassInfo = name.getClass();  
        System.out.println("Parent is: " + nameClassInfo.getSuperclass());  
    }  
}
```

- The names of these classes would have been stored in the constant pool for TestClassClass.
- In this phase, the **names are replaced with their actual references**.

Class Initialization

- This is the process of **setting** class variables to their **proper initial values** - initial values desired by the programmer.

```
class Example1 {  
    static double rate = 3.5;  
    static int size = 3*(int)(Math.random()*5);  
    ...  
}
```

- Initialization of a class consists of two steps:
 - Initializing its direct super class (if any and if not already initialized)
 - Executing its own initialization statements
- The above imply that, the first class that gets initialized is **Object**.
- Note that **static final variables** are not treated as class variables but as constants and are assigned their values at compilation.

```
class Example2 {  
    static final int angle = 35;  
    static final int length = angle * 2;  
    ...  
}
```

Class Area

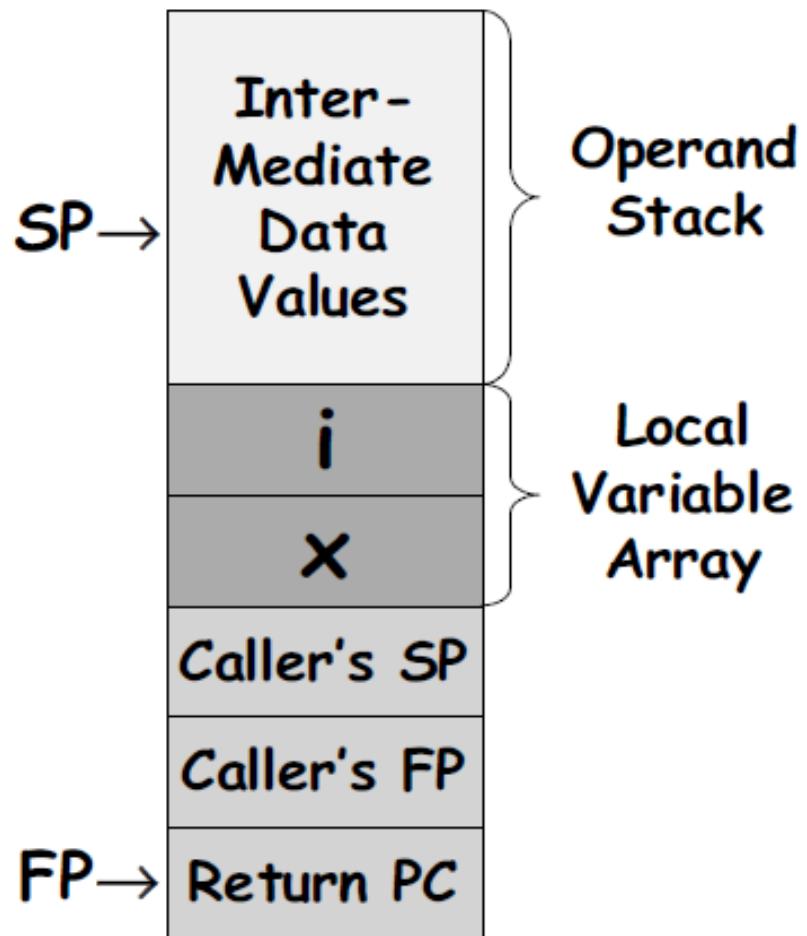
- Class Information
 - Internal representation of Java classes
 - Information about the superclass and implemented interfaces
 - Information about the fields and methods
- Constant Pool
- Method Area
 - Contains the bytecodes of the methods of the loaded classes

Stack

- The Java stack is composed of frames
 - A frame contains the state of one Java method invocation
 - Logically, a frame has two parts: local variable array and operand stack
- JVM has no registers; it uses the operand stack for storage of intermediate data values
 - to keep the JVM's instruction set compact
 - to facilitate implementation on architectures with limited number of general purpose registers
- Each Java thread has its own stack and cannot access the stack of other threads

Stack Frame

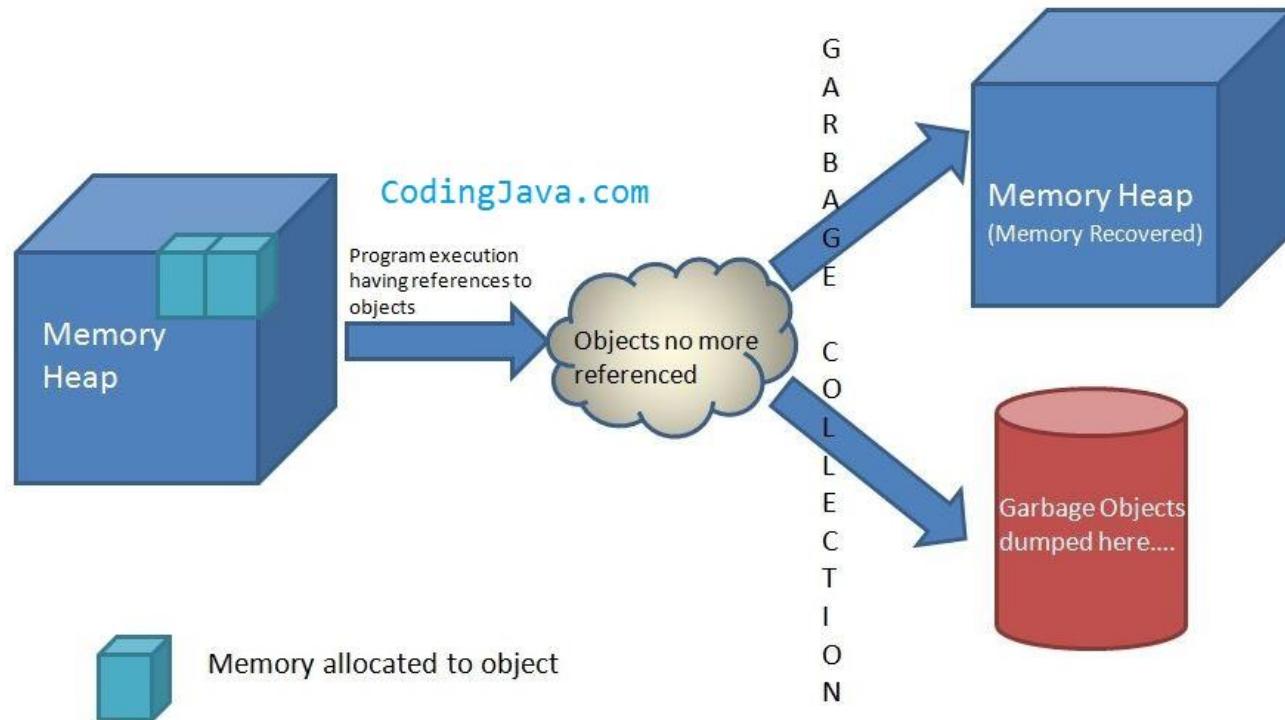
```
public class A
{
    ...
    void f(int x)
    {
        int i;
        for(i=0; i<x; i++)
        {
            ...
        }
    ...
}
```



Heap

- All Java objects are allocated in the heap
- Java applications cannot explicitly free an object
- The Garbage Collector is invoked from time to time automatically to reclaim the objects that are no longer needed by the application
- The heap is shared by all Java threads

Garbage Collection



How Objects are Created in Java

- An object is created in Java by invoking the new() operator.
- Calling the new() operator, the JVM will do the following:
 - **allocate memory;**
 - **assign fields their default values;**
 - **run the constructor;**
 - **a reference is returned.**

How Java Reclaims Objects Memory

- Java does not provide the programmer any means to destroy objects explicitly
- The advantages are
 - No *dangling reference* problem in Java
 - Easier programming
 - No *memory leak* problem

What is Garbage?

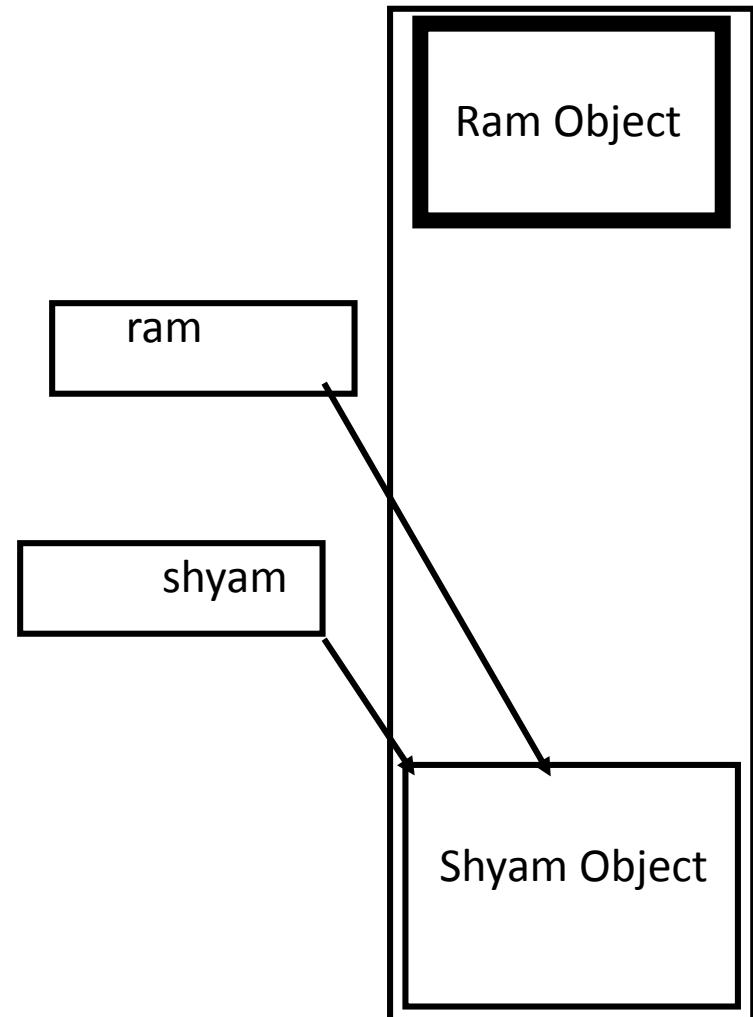
Garbage: unreferenced objects

```
Student ram= new Student();
```

```
Student shyam= new Student();
```

```
ram=shyam;
```

*Now Ram Object becomes a garbage,
It is unreferenced Object*



Garbage Collection:

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically.

Advantage of Garbage Collection:

- It makes java memory efficient because garbage collector removes the unreferenced objects from heap memory.
- It is automatically done by the garbage collector so we don't need to make extra efforts.

How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

What is Garbage Collection?

- *What is Garbage Collection?*
 - Finding garbage and reclaiming memory allocated to it.
- *Why Garbage Collection?*
 - the heap space occupied by an un-referenced object can be recycled and made available for subsequent new objects
- *When is the Garbage Collection process invoked?*
 - When the total memory allocated to a Java program exceeds some threshold.
- *Is a running program affected by garbage collection?*
 - Yes, the program suspended during garbage collection.

Advantages of Garbage Collection

- GC eliminates the need for the programmer to deallocate memory blocks explicitly
- Garbage collection helps ensure program integrity.
- Garbage collection can also dramatically simplify programs.

Disadvantages of Garbage Collection

- Garbage collection adds an overhead that can affect program performance.
- GC requires extra memory.
- Programmers have less control over the scheduling of CPU time.

finalize() method:

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in System class as:

[copy to clipboard](#)

```
protected void finalize(){}
```

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method:

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

[copy to clipboard](#)

```
public static void gc(){}
```

Simple Example of garbage collection:

```
class Simple{  
  
    public void finalize(){System.out.println("object is garbage collected");}  
  
    public static void main(String args[]){  
        Simple s1=new Simple();  
        Simple s2=new Simple();  
        s1=null;  
        s2=null;  
        System.gc();  
    }  
}
```

Output:
object is garbage collected
object is garbage collected

GarbageDemo.java

Garbage Collector

```
public class GCDemo {  
  
    public void finalize(){  
        System.out.println("garbage collector invoked!!");  
    }  
  
    public static void main(String[] args) {  
        GCDemo[] gc=new GCDemo[10];  
        for(int i=0;i<10;i++){  
            gc[i]=new GCDemo();  
        }  
        gc=null;  
        System.gc();  
    }  
}
```

Common Garbage Collection Schemes

- Three main approaches of garbage collection:
 - Reference counting
 - Mark-and-sweep
 - Stop-and-copy garbage collection.

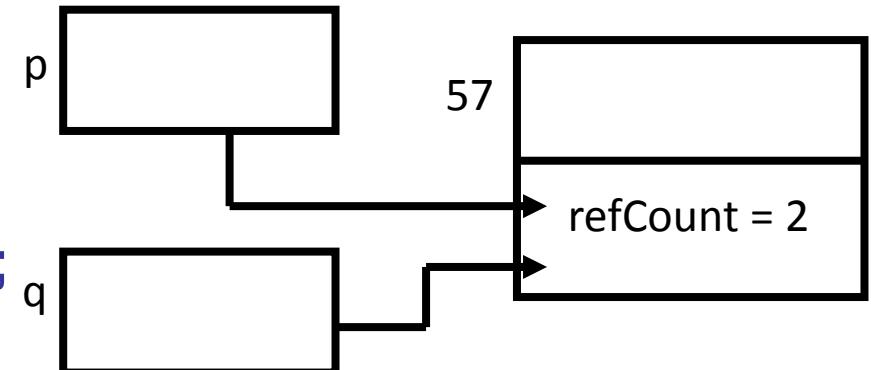
Reference Counting Garbage Collection

- Main Idea: Add a reference count field for every object.
- This field is updated when the number of references to an object changes.

Example

`Object p= new Integer(57);`

`Object q = p;`

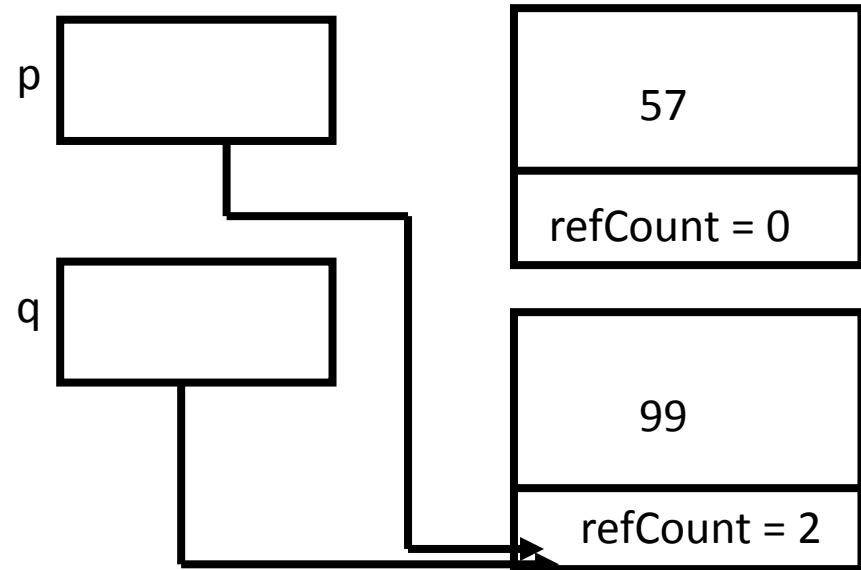


Reference Counting (cont'd)

- The update of reference field when we have a reference assignment (i.e $p=q$) can be implemented as follows

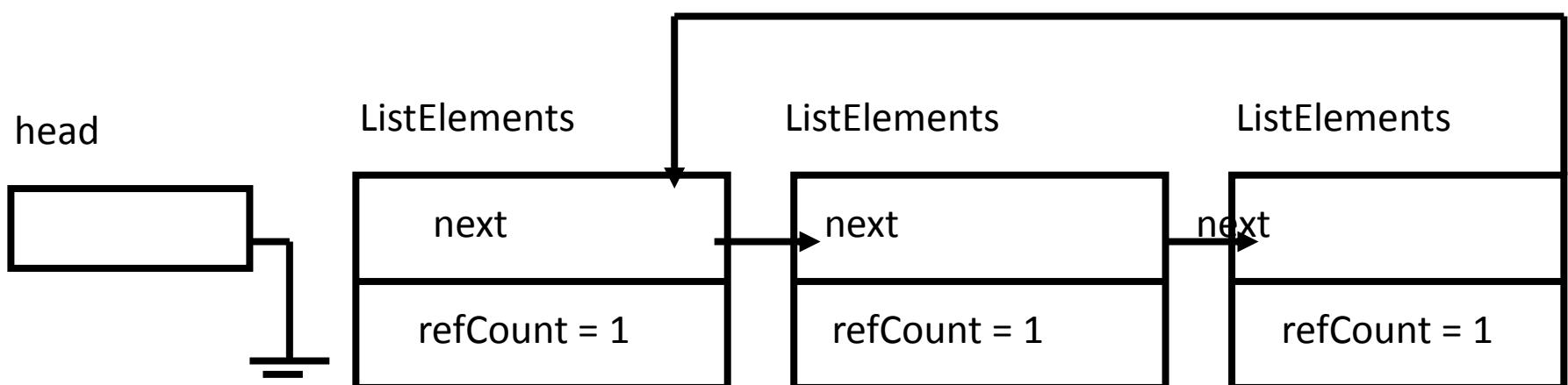
Example:

```
Object p = new Integer(57);
Object q= new Integer(99);
p=q
```



Reference Counting (cont'd)

- Reference counting will fail whenever the data structure contains a cycle of references and the cycle is not reachable from a global or local reference*



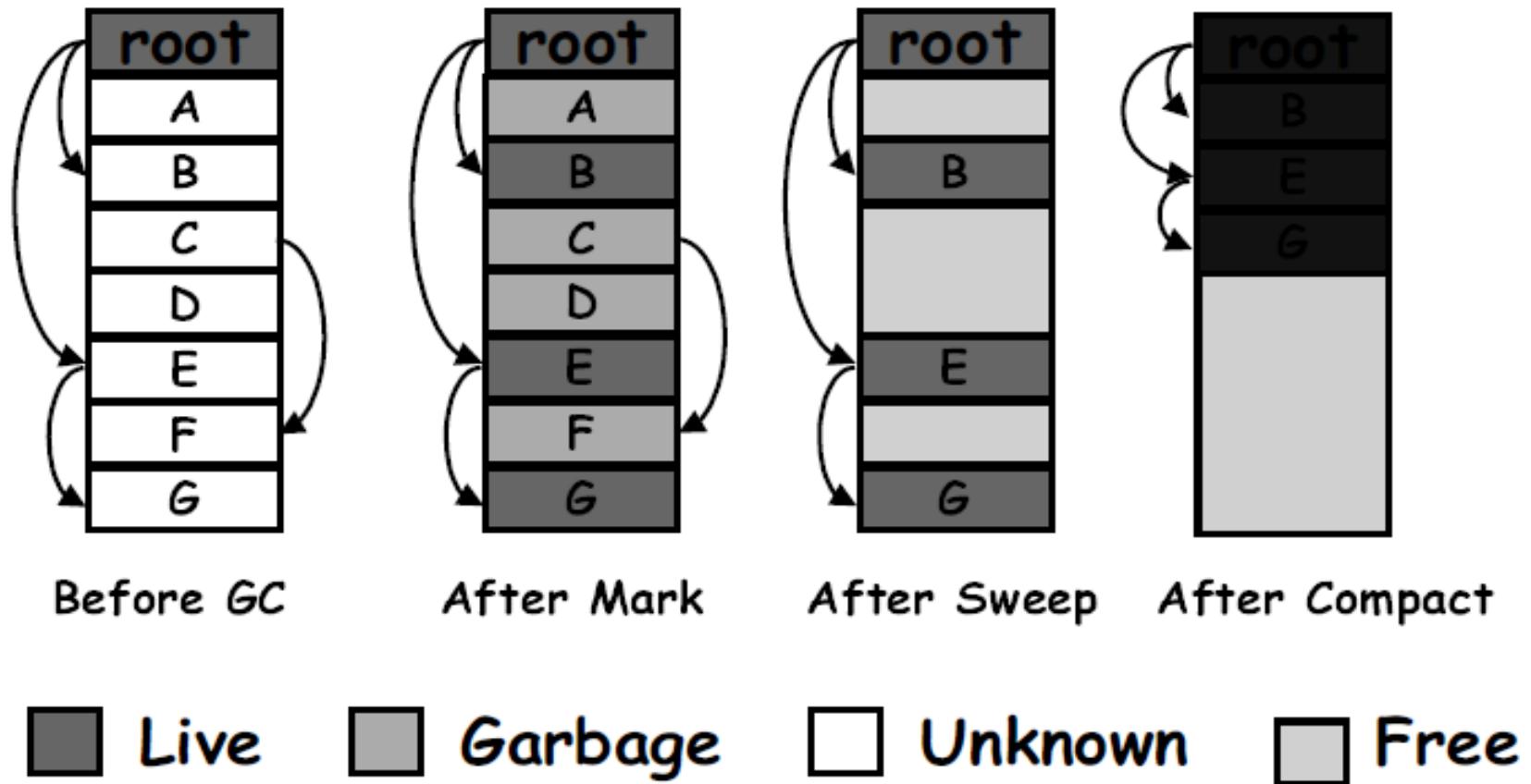
Reference Counting (cont'd)

- Advantages
 - Conceptually simple: Garbage is easily identified
 - It is easy to implement.
 - Immediate reclamation of storage
 - Objects are not moved in memory during garbage collection.
- Disadvantages
 - Reference counting does not detect garbage with cyclic references.
 - The overhead of incrementing and decrementing the reference count each time.
 - Extra space: A count field is needed in each object.
 - It may increase heap fragmentation.

Mark-and-Sweep Garbage Collection

- The **mark-and-sweep** algorithm is divided into two phases:
 - **Mark phase:** the garbage collector traverses the graph of references from the *root nodes* and marks each heap object it encounters. Each object has an extra bit: the mark bit – initially the mark bit is 0. It is set to 1 for the *reachable objects* in the mark phase.
 - **Sweep phase:** the GC scans the heap looking for objects with mark bit 0 – these objects have not been visited in the mark phase – they are garbage. Any such object is added to the free list of objects that can be reallocated. The objects with a mark bit 1 have their mark bit reset to 0.

Mark / Sweep / Compaction



Mark and Sweep (cont'd)

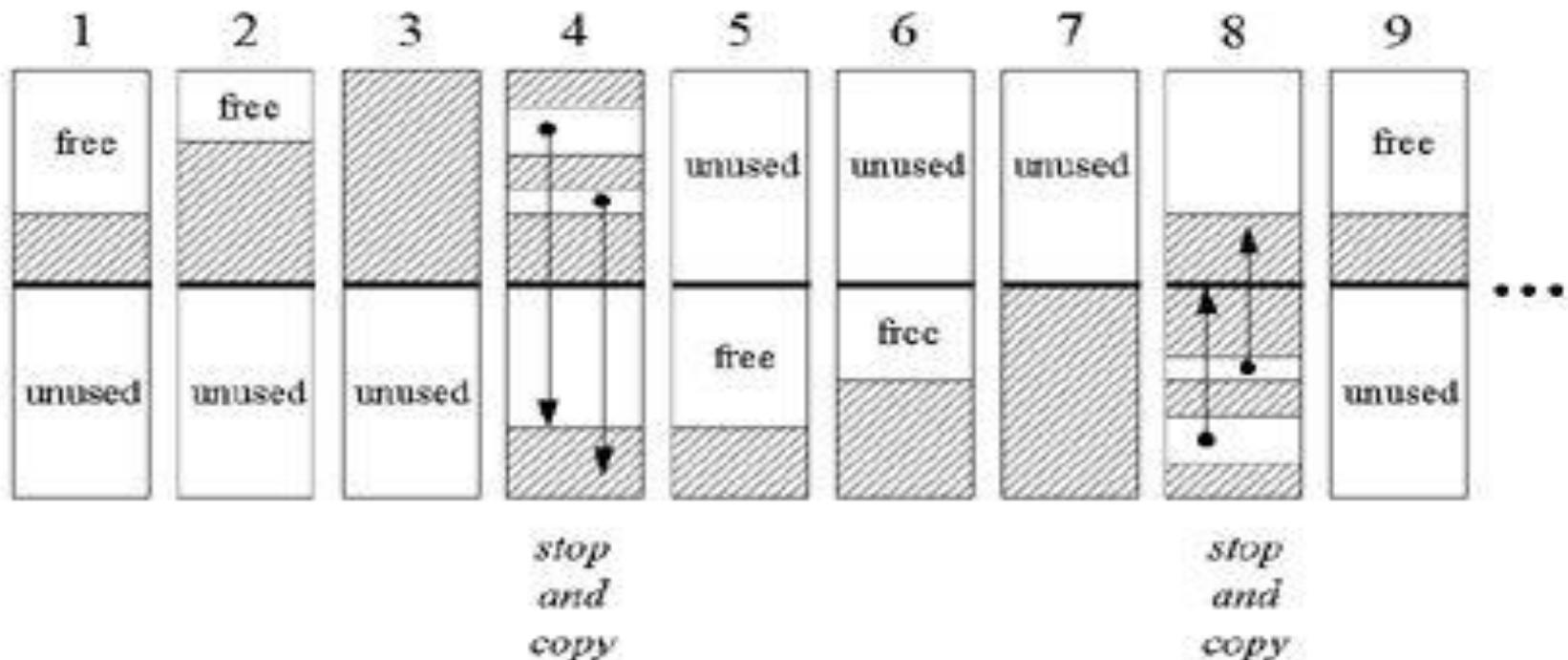
- Advantages
 - It is able to reclaim garbage that contains cyclic references.
 - There is no overhead in storing and manipulating reference count fields.
 - Objects are not moved during GC – no need to update the references to objects.
- Disadvantages
 - It may increase heap fragmentation.
 - It does work proportional to the size of the entire heap.
 - The program must be halted while garbage collection is being performed.

Stop-and-Copy Garbage Collection

- The heap is divided into two regions: Active and Inactive.
- Objects are allocated from the **active** region only.
- When all the space in the active region has been exhausted, program execution is stopped and the heap is traversed. Live objects are copied to the other region as they are encountered by the traversal. The role of the two regions is reversed, i.e., swap (active, inactive). ...

Stop-and-Copy Garbage Collection (cont'd)

- A graphical depiction of a garbage-collected heap that uses a stop and copy algorithm. This figure shows nine snapshots of the heap over time:



Stop-and-Copy Garbage Collection (cont'd)

- Advantages
 - Only one pass through the data is required.
 - It de-fragments the heap.
 - It does work proportional to the amount of live objects and not to the memory size.
 - It is able to reclaim garbage that contains cyclic references.
 - There is no overhead in storing and manipulating reference count fields.

Stop-and-Copy Garbage Collection (cont'd)

- Disadvantages
 - Twice as much memory is needed for a given amount of heap space.
 - Objects are moved in memory during garbage collection (i.e., references need to be updated)
 - The program must be halted while garbage collection is being performed.

Methods of Object class

The Object class provides many methods. They are as follows:

1. **public final Class getClass()** returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
2. **public int hashCode()** returns the hashcode number for this object.
3. **public boolean equals(Object obj)** compares the given object to this object.
4. **protected Object clone() throws CloneNotSupportedException** creates and returns the exact copy (clone) of this object.
5. **public String toString()** returns the string representation of this object.
6. **public final void notify()** wakes up single thread, waiting on this object's monitor.
7. **public final void notifyAll()** wakes up all the threads, waiting on this object's monitor.
8. **public final void wait(long timeout) throws InterruptedException** causes the current thread to wait for the specified miliseconds, until another thread notifies (invokes notify() or notifyAll() method).
9. **public final void wait(long timeout,int nanos) throws InterruptedException** causes the current thread to wait for the specified miliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
10. **public final void wait() throws InterruptedException** causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
11. **protected void finalize() throws Throwable** is invoked by the garbage collector before object is being garbage collected.

Class “Object” : getClass()

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ObjectTest {
    public static void main(String[] args) {
        int count=0;
        Object o=new String("CDAC Hyderabad");
        Class c=o.getClass();
        System.out.println("FQN of class:"+c.getName());
        Method[] m=c.getDeclaredMethods(); //reflection
        Field[] f=c.getDeclaredFields(); // reflection
        for(Method m1:m){
            count++;
            System.out.println(m1.getName());
        }
        System.out.println("No of methods:"+count);
        System.out.println(".....");
        for(Field f1:f){
            count++;
            System.out.println(f1.getName());
        }
    }
}
```

Output:

FQN of	getBytes
class:java.lang.String	getBytes
equals	getBytes
toString	getBytes
hashCode	getChars
compareTo	getChars
compareTo	indexOfSupplementary
indexOf	intern
indexOf	isEmpty
indexOf	join
indexOf	join
indexOf	lastIndexOf
valueOf	length
valueOf	matches
valueOf	nonSyncContentEquals
valueOf	offsetByCodePoints
valueOf	regionMatches
valueOf	regionMatches
valueOf	replace
valueOf	replace
valueOf	replaceAll
valueOf	replaceFirst
charAt	split
checkBounds	split
codePointAt	startsWith
codePointBefore	startsWith
codePointCount	subSequence
compareIgnoreCase	substring
concat	toCharArray
contains	toLowerCase
contentEquals	toLowerCase
contentEquals	toUpperCase
copyValueOf	toUpperCase
copyValueOf	trim
endsWith	No of methods:77
equalsIgnoreCase
format	value
format	hash
	serialVersionUID
	serialPersistentFields
	CASE_INSENSITIVE_ORDER

Coming session

Inheritance

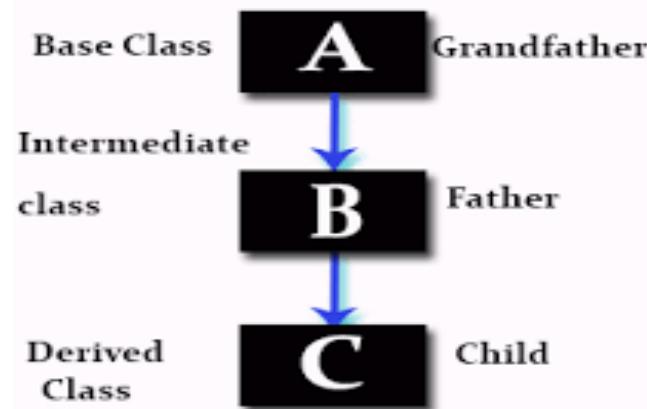


Fig: Multilevel Inheritance

Session 4:

Learning Objectives

By the end of this session, you must be able to

- **Describe Inheritance**
- **Use Super Keyword**
- **Explain Nested Classes**

Inheritance

JAVA INHERITANCE 13 - BY KSIMPLIFIED

WWW.TOONDOO.COM

How about writing Java classes
for this inheritance hierarchy?



Animal Kingdom

Mammals

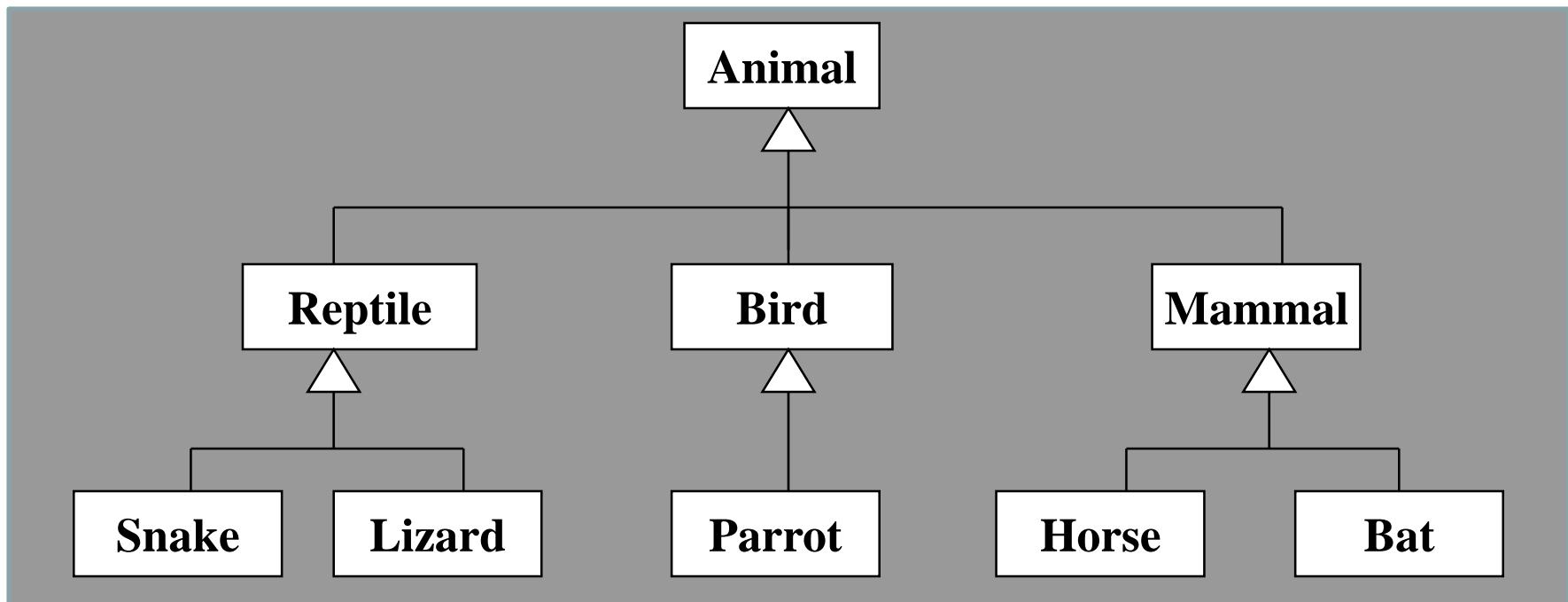


Inheritance

- **INHERITANCE** is One of the pillars of object-orientation
- A new class is derived from an existing class:
 - 1) Existing class is called **super-class**
 - 2) Derived class is called **sub-class**
- A sub-class is a specialized version of its super-class:
 - 1) has **all non-private** members of its super-class
 - 2) may provide its **own implementation** of super-class methods
- **Objects** of a **sub-class** are a special kind of **objects** of a **super-class**

Class Hierarchy

- A child class of one parent can be the parent of another child, forming ***class hierarchies***



extends Key Word

- It is a keyword used to inherit a class from another class

```
class One  
{  
    int a=5;  
}
```

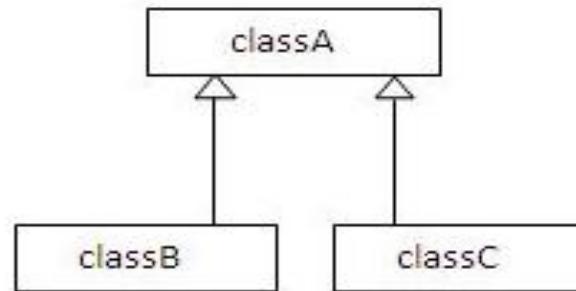
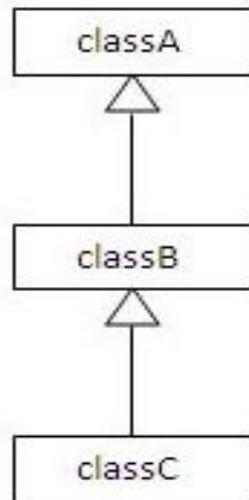
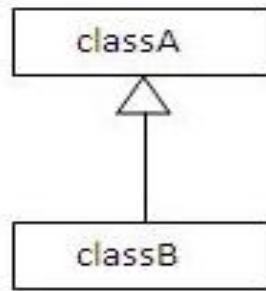
```
class Two extends One  
{  
    int b=10;  
}
```

One baseobj=new One(); // base class object.
Two subobj=new Two(); // child class object

super class object **baseobj** can be used to refer its sub class objects.

For example,
Baseobj=subobj // now its pointing to sub class

Types of Inheritance:



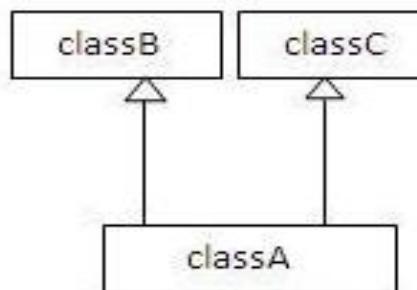
1)Single

2)Multilevel

3)Hierarchical

Note: Multiple inheritance is not supported in java in case of class.

When a class extends two classes i.e. known as multiple inheritance. For Example:



Example : Inheritance

```
class Employee{  
    int salary=40000;  
  
}  
  
class Programmer extends Employee{  
    int bonus=10000;  
  
    Public Static void main(String args[]){  
        Programmer p=new Programmer();  
        System.out.println("Programmer salary is:"+p.salary);  
        System.out.println("Bonus of Programmer is:"+p.bonus);  
    }  
}
```

Developer.java

Overriding

Allows a **sub class or child class** to provide a specific implementation of a method that is already provided by one of its **super classes or parent classes**.

Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method is same and there is IS-A relationship between the classes, so there is method overriding.

```
//<b>Example of method overriding</b>

class Vehicle{
void run(){System.out.println("Vehicle is running");}
}

class Bike extends Vehicle{
void run(){System.out.println("Bike is running safely");}
}

public static void main(String args[]){
Bike obj = new Bike();
obj.run();
}
}
```

Output:Bike is running safely

Bike.java

Difference between method Overloading and Method Overriding.

There are three basic differences between the method overloading and method overriding. They are as follows:

Method Overloading	Method Overriding
1) Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2) method overlaoding is performed within a class.	Method overriding occurs in two classes that have IS-A relationship.
3) In case of method overloading parameter must be different.	In case of method overriding parameter must be same.

The Benefits of Inheritance

- Software Reusability (among projects)
- Increased Reliability (resulting from reuse and sharing of well-tested code)
- Code Sharing (within a project)
- Consistency of Interface (among related objects)
- Software Components
- Rapid Prototyping (quickly assemble from pre-existing components)
- Polymorphism and Frameworks (high-level reusable components)
- Information Hiding

The Costs of Inheritance

- Execution Speed
- Program Size
- Message-Passing Overhead
- Program Complexity (in overuse of inheritance)

super keyword:

super is a reference variable that is used to refer immediate parent class object.

Uses of super Keyword:

1. super is used to refer immediate parent class instance variable.
 2. super() is used to invoke immediate parent class constructor.
 3. super is used to invoke immediate parent class method.
-
-

1 . Referring immediate parent class instance variable

Problem without super keyword

```
class Vehicle{  
    int speed=50;  
}  
  
class Bike extends Vehicle{  
    int speed=100;  
  
    void display(){  
        System.out.println(speed); //will print speed of Bike  
    }  
  
    public static void main(String args[]){  
        Bike b=new Bike();  
        b.display();  
  
    }  
}
```

Solution by super keyword

```
<b><i>/example of super keyword</i></b>  
  
class Vehicle{  
    int speed=50;  
}  
  
class Bike extends Vehicle{  
    int speed=100;  
  
    void display(){  
        System.out.println(super.speed); //will print speed of Bike  
    }  
  
    public static void main(String args[]){  
        Bike b=new Bike();  
        b.display();  
  
    }  
}
```

SuperBike.java

2.super is used to invoke parent class constructor.

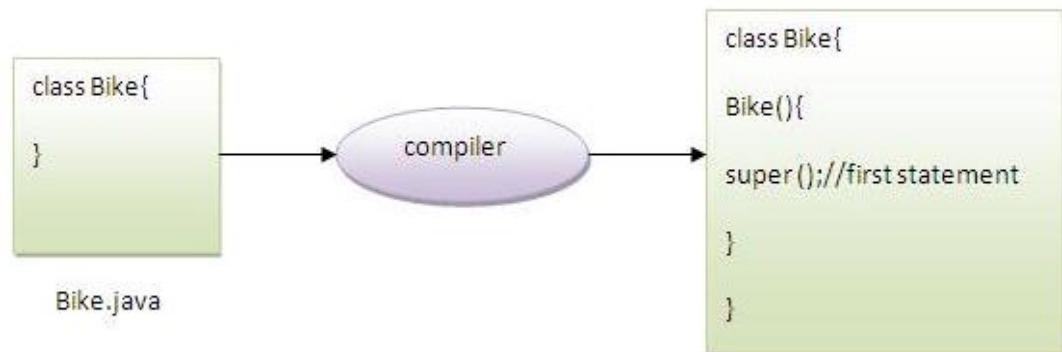
The super keyword can also be used to invoke the parent class constructor as given below:

Example of super keyword

```
class Vehicle{  
    Vehicle(){System.out.println("Vehicle is created");}  
}  
  
class Bike extends Vehicle{  
    Bike(){  
        super(); //will invoke parent class constructor  
        System.out.println("Bike is created");  
    }  
    public static void main(String args[]){  
        Bike b=new Bike();  
    }  
}
```

Note:super() is added in each class constructor automatically by compiler.

Output:Vehicle is created
Bike is created



Bike.class

Program of super that can be used to invoke method of parent class

```
class Person{
void message(){System.out.println("welcome");}
}

class Student extends Person{
void message(){System.out.println("welcome to java");}

void display(){
message(); //will invoke current class message() method
super.message(); //will invoke parent class message() method
}

public static void main(String args[]){
Student s=new Student();
s.display();
}
}
```

Output:welcome to java
welcome

final keyword:

The final keyword in java is used to restrict the user.
The final keyword can be used in many context. Final
can be:

1. variable
2. method
3. class

1. Final variable can not be **changed**
2. Final method can not be **overridden**
3. Final class can not be **inherited**

Class “Object”

- In Java, all classes use inheritance.
- If no parent class is specified explicitly, the base class **Object** is implicitly inherited.
- All classes defined in Java, are children of **Object** class, which provides minimal functionality guaranteed common to all objects.

Class Object

Methods defined in Object class are;

- 1.equals(Object obj)** Determine whether the argument object is the same as the receiver
- 2.getClass()** Returns the class of the receiver, an object of type Class
- 3.hashCode()** Returns a hash value for this object. Should be overridden when the equals method is changed
- 4.toString()** Converts object into a string value. This method is also often overridden
- 5.finalize()** Called by the garbage collector on an object when garbage collector determines that there are no more references to the object
- 6.clone()** Creates and returns a copy of this object.
- 7.wait()**
- 8.wait(long timeout)**
- 9.wait(long timeout, int nanos)**
- 10. notify()**
- 11. notifyAll()**

Class “Object” : getClass()

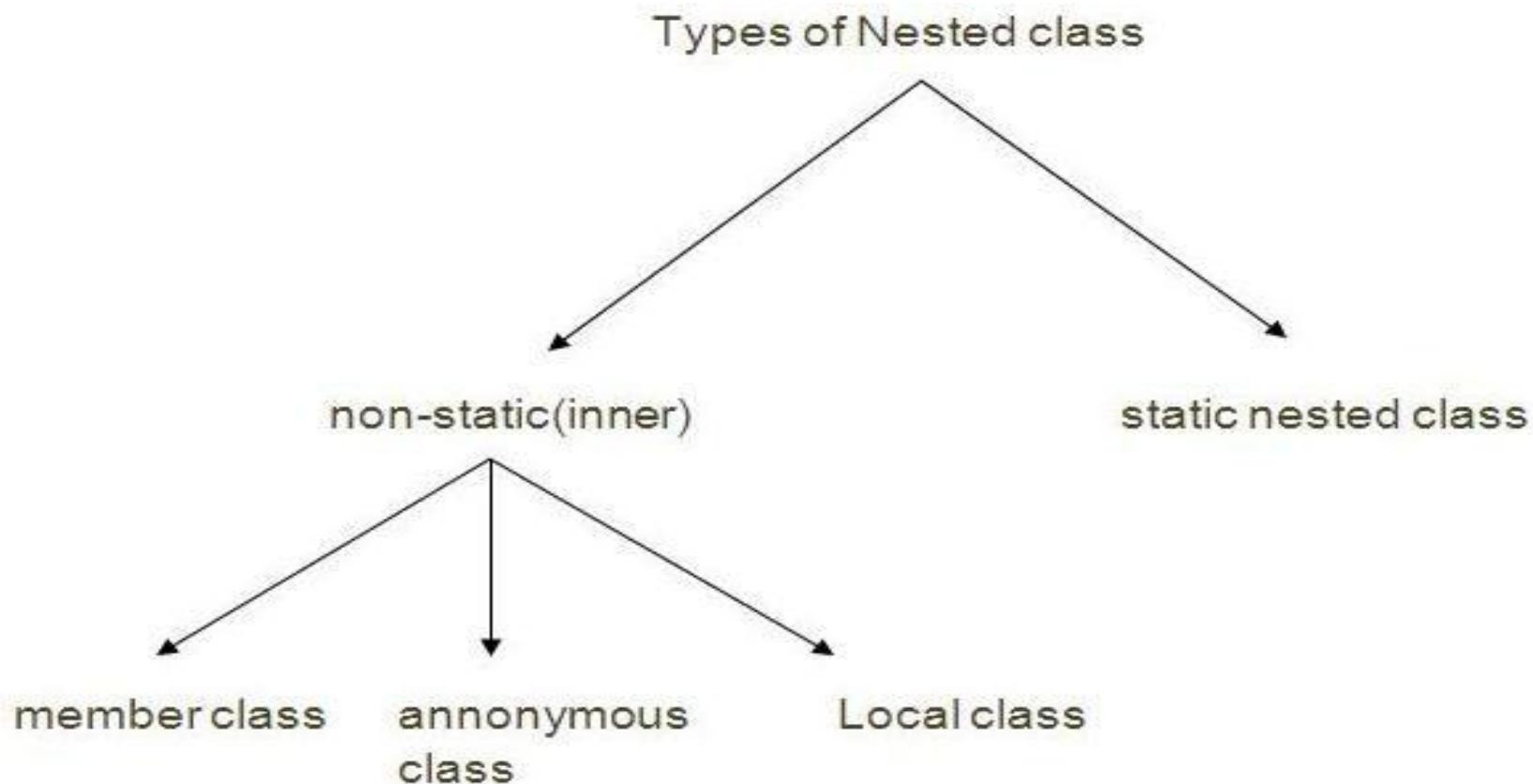
```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ObjectTest {
    public static void main(String[] args) {
        int count=0;
        Object o=new String("CDAC Hyderabad");
        Class c=o.getClass();
        System.out.println("FQN of class:"+c.getName());
        Method[] m=c.getDeclaredMethods(); //reflection
        Field[] f=c.getDeclaredFields(); // reflection
        for(Method m1:m){
            count++;
            System.out.println(m1.getName());
        }
        System.out.println("No of methods:"+count);
        System.out.println(".....");
        for(Field f1:f){
            count++;
            System.out.println(f1.getName());
        }
    }
}
```

Output:

FQN of	getBytes
class:java.lang.String	getBytes
equals	getBytes
toString	getBytes
hashCode	getChars
compareTo	getChars
compareTo	indexOfSupplementary
indexOf	intern
indexOf	isEmpty
indexOf	join
indexOf	join
indexOf	lastIndexOf
valueOf	length
valueOf	matches
valueOf	nonSyncContentEquals
valueOf	offsetByCodePoints
valueOf	regionMatches
valueOf	regionMatches
valueOf	replace
valueOf	replace
valueOf	replaceAll
valueOf	replaceFirst
charAt	split
checkBounds	split
codePointAt	startsWith
codePointBefore	startsWith
codePointCount	subSequence
compareIgnoreCase	substring
concat	toCharArray
contains	toLowerCase
contentEquals	toLowerCase
contentEquals	toUpperCase
copyValueOf	toUpperCase
copyValueOf	trim
endsWith	No of methods:77
equalsIgnoreCase
format	value
format	hash
	serialVersionUID
	serialPersistentFields
	CASE_INSENSITIVE_ORDER

Nested Classes



Nested Classes

- The Java programming language allows you to define a class within another class.
Such a class is called a ***nested class***
- Nested classes introduced in jdk1.1

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

Nested Classes - Terminology

- Nested classes are divided into **two** categories:
 - **static**
Nested classes that are declared **static** are simply called ***static nested classes***
 - **non-static**
Non-static nested classes are called ***inner classes***

```
class OuterClass {  
    ...  
    static class StaticNestedClass {  
        ...  
    }  
    class InnerClass {  
        ...  
    }  
}
```

What is Nested Class?

- A nested class is a member of its enclosing class
- Non-static nested classes (**inner classes**) have access to other members of the enclosing class, even if they are declared **private**
- **Static nested classes** do not have access to other members of the enclosing class except **static members**
- As a member of the OuterClass, a **nested class** can be declared **private, public, protected, or static**

Why Use Nested Classes?

- There are several compelling reasons for using nested classes, among them are :
 - It is a way of **logically grouping** classes that are only used in one place
 - It **increases encapsulation**
 - Nested classes can lead to **more readable and maintainable code (less code)**

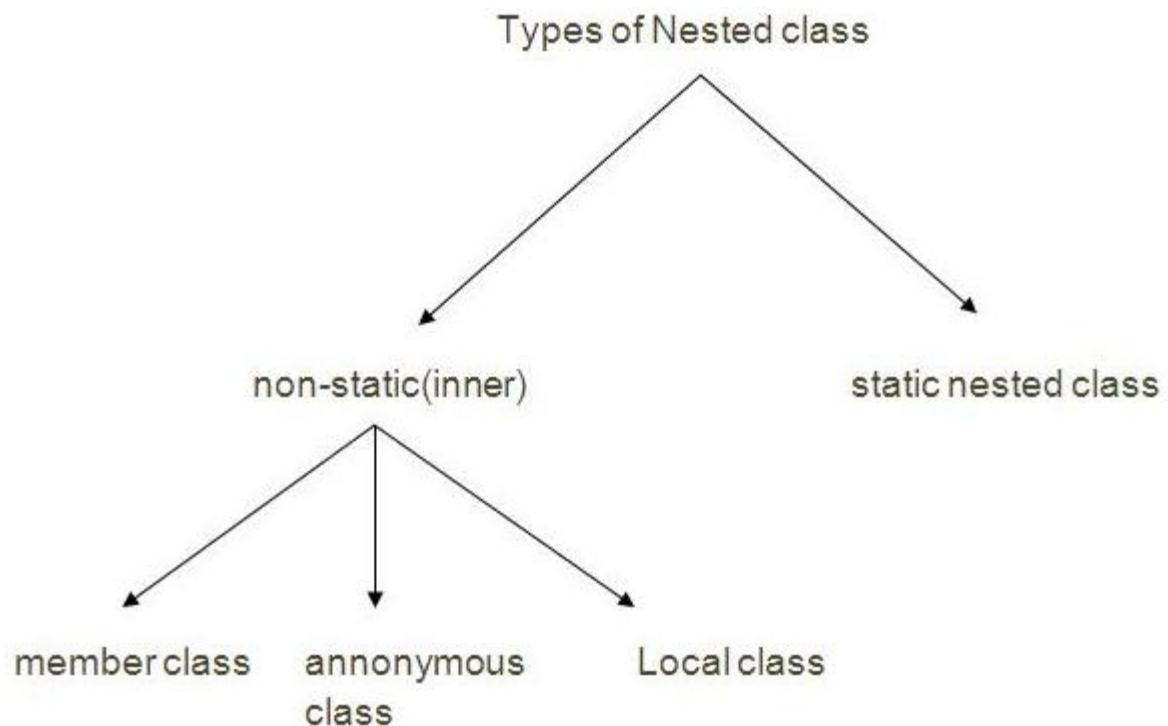
Types of nested classes

- **Static nested classes**

- Non-local named only

- **Inner classes**

- Local
 - *Anonymous or named*
- Non-local
 - *Named only*

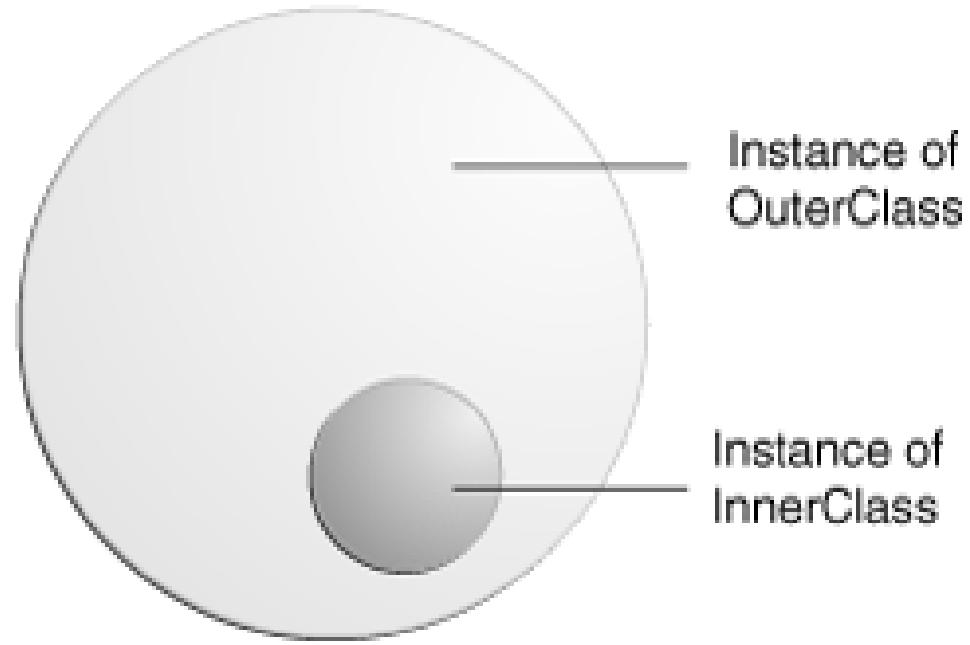


Inner Classes

- As with **instance methods and variables**, an **inner class** is associated with an instance of its enclosing class and has direct access to that object's methods and fields
- Also, 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**.

Inner 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

An Instance of `InnerClass` Exists Within an
Instance of `OuterClass`

Inner class

- To instantiate an **inner class**, you must first instantiate the outer class.
- Then, create the inner object within the outer object with this syntax:

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

The .class File for an Inner Class

- Compiling any class in Java produces a **.class** file named **ClassName.class**
- Compiling a class with one (or more) **inner classes** causes both (or more) classes to be compiled, and produces two (or more) **.class** files
 - Such as
 - `ClassName.class` and
 - `ClassName$InnerClassName.class`

Example of member inner class that is invoked inside a class

In this example, we are invoking the method of member inner class from the display method of Outer class.

```
class Outer{
    private int data=30;
    class Inner{
        void msg(){System.out.println("data is "+data);}
    }

    void display(){
        Inner in=new Inner();
        in.msg();
    }

    public static void main(String args[]){
        Outer obj=new Outer();
        obj.display();
    }
}
```

Output: data is 30

[Outer.java](#)

Example of member inner class that is invoked outside a class

In this example, we are invoking the msg() method of Inner class from outside the outer class i.e. Test class.

//Program of member inner class that is invoked outside a class

```
class Outer{  
    private int data=30;  
    class Inner{  
        void msg(){System.out.println("data is"+data);}  
    }  
}  
  
class Test{  
    public static void main(String args[]){  
        Outer obj=new Outer();  
        Outer.Inner in=obj.new Inner();  
        in.msg();  
    }  
}
```

Output: data is 30

[OuterTest.java](#)

Local Class

- A class that is created inside a method
- Local class can not have static members.
- Final static fields are allowed
- Can not have accessibility modifiers.

Program of local inner class

```
class Simple{  
    private int data=30;//instance variable  
    void display(){  
        class Local{  
            void msg(){System.out.println(data);}  
        }  
        Local l=new Local();  
        l.msg();  
    }  
    public static void main(String args[]){  
        Simple obj=new Simple();  
        obj.display();  
    }  
}
```

LocalDemo.java

Output : 30

Anonymous Class

- Combines the process of **definition and instantiation** into a single step
- **Syntax:**

```
new <Super Class Name>(<optional arg list>
{
    <member declaration>
};
```

Anonymous Class can be created by:

1. Class (May be **abstract** class)
2. Interface

Anonymous Inner Class

```
@FunctionalInterface  
interface B{  
void show();  
}  
  
public class A {  
public static void main(String[] args){  
    B b=new B(){  
        public void show(){  
System.out.println("Anonymous  
Implementation");  
        }  
    };  
    b.show();  
}  
}  
  
// Anonymous Implementation
```

```
@FunctionalInterface  
interface B{  
void show();  
}  
  
public class A {  
public static void main(String[] args){  
B b=()->System.out.println("Anonymous  
Implementation"); // lambda expression  
    b.show();  
}  
}
```

Program of anonymous inner class by abstract class

```
abstract class Person{
    abstract void eat();
}

class Emp{
    public static void main(String args[]){
        Person p=new Person(){
            void eat(){System.out.println("nice fruits");}
        };

        p.eat();
    }
}
```

Output:nice fruits

[Emp.java](#)

What happens behind this code?

```
Person p=new Person(){  
    void eat(){System.out.println("nice fruits");}  
};  
}  
}
```

1. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Person type. As you know well that Parent class reference variable can refer the object of Child class.

Static Nested Classes

- As with class methods and variables, a static nested class is associated with its outer class
- And like static class methods, 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**
- A static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class

```
class Cover{  
    static class InnerCover{  
        void go(){  
            System.out.println("I am the first Static Inner  
Class");  
        }  
    }  
}
```

Program of static nested class that have instance method

```
class Outer{
    static int data=30;

    static class Inner{
        void msg(){System.out.println("data is "+data);}
    }

    public static void main(String args[]){
        Outer.Inner obj=new Outer.Inner();
        obj.msg();
    }
}
```

Output: data is 30

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

StaticClassDemo.java

Program of static nested class that have static method

```
class Outer{
    static int data=30;

    static class Inner{
        static void msg(){System.out.println("data is "+data);}
    }

    public static void main(String args[]){
        Outer.Inner.msg(); //no need to create the instance of static nested class
    }
}
```

Output: data is 30

[StaticClassDemo1.java](#)

Questions?

Thank You,
Sadhu Sreenivas

Next...

- **Abstract Class**
- **Interfaces**
- **Packages**
- **Access Modifiers**
- **Wrapper Classes**

Session 5:

Learning Objectives

By the end of this session, you must be able to

- **Explain Abstract Class**
- **Describe Interfaces**
- **Create and Use Packages**
- **Explain Wrapper Classes**

Abstraction

Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only important things to the user and hides the internal details. Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

Syntax to declare the abstract class

```
abstract class <class_name>{}
```

abstract method

A method that is declared as abstract and does not have implementation is known as abstract method.

Syntax to define the abstract method

```
abstract return_type <method_name>(); //no braces{}
```

Example of abstract class that have abstract method

In this example, Bike the abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{
    abstract void run();
}

class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}

public static void main(String args[]){
    Bike obj = new Honda();
    obj.run();
}
```

[Honda.java](#)

abstract class - features

- If a class contains at least one **abstract** method should be declared as **abstract**
- We can declare a class as **abstract**, even if the class does not have any **abstract** methods
- **Abstract** class can not be instantiated but can be referred
- If a class is extending from an **abstract** class, the extending class should provide the body (implementation) for all the **abstract** methods of super class
- If the extended class fails to provide body for at least one **abstract** method, should be declared **abstract**

Example

```
abstract class Shape{
abstract void draw();
}

class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}

class Circle extends Shape{
void draw(){System.out.println("drawing circle");}
}

class Test{
public static void main(String args[]){
Shape s=new Circle();
//In real scenario, Object is provided through factory method
s.draw();
}
}
```

Output: drawing circle

Example

//<i>example of abstract class that have method body</i>

```
abstract class Bike{
    abstract void run();
    void changeGear(){System.out.println("gear changed");}
}

class Honda extends Bike{
    void run(){System.out.println("running safely..");}
}

public static void main(String args[]){
    Bike obj = new Honda();
    obj.run();
    obj.changeGear();
}
}
```

Output:running safely..
 gear changed

Example

```
//<b><i>example of abstract class having constructor, field and method</i></b>
abstract class Bike
{
    int limit=30;
    Bike(){System.out.println("constructor is invoked");}
    void getDetails(){System.out.println("it has two wheels");}
    abstract void run();
}

class Honda extends Bike{
    void run(){System.out.println("running safely..");}

    public static void main(String args[]){
        Bike obj = new Honda();
        obj.run();
        obj.getDetails();
        System.out.println(obj.limit);
    }
}
```

Output:constructor is invoked
running safely..
it has two wheels
30

Interfaces

- An *interface* is a reference type, similar to a class, that can contain **only constants, method signatures, and nested types**.
- There are **no method bodies**.
- Interfaces cannot be instantiated—they can only be **implemented** by classes or **extended** by other interfaces.
- A method declaration within an interface is followed by a semicolon, but no braces.
- All methods declared in an interface are implicitly **public**.
- An interface can contain **constant declarations** in addition to **method declarations**.
- All constant values defined in an interface are implicitly **public static final**.

Uses of Interfaces

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Simple example of Interface

In this example, Printable interface have only one method, its implementation is provided in the A class.

```
interface printable{
    void print();
}

class A implements printable{
    public void print(){System.out.println("Hello");}

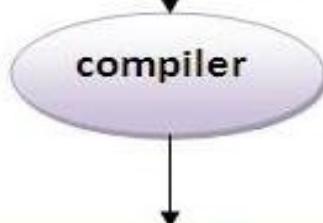
    public static void main(String args[]){
        A obj = new A();
        obj.print();
    }
}
```

Output:Hello

The java compiler converts methods of interface as public and abstract, data members as public,final and static by default.

```
interface Printable{  
    int MIN=5;  
    void print();  
}
```

Printable.java



```
interface Printable{  
    public static final int MIN=5;  
    public abstract void print();  
}
```

Multiple inheritance by interface

A class cannot extend two classes but it can implement two interfaces. For example:

```
interface printable{
    void print();
}

interface Showable{
    void show();
}

class A implements printable,Showable{

    public void print(){System.out.println("Hello");}
    public void show(){System.out.println("Welcome");}

    public static void main(String args[]){
        A obj = new A();
        obj.print();
        obj.show();
    }
}
```

A.java

A class implements interface but One interface extends another interface .

```
interface Printable{
void print();
}

interface Showable extends Printable{
void show();
}

class A implements Showable{

public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome")}

public static void main(String args[]){
A obj = new A();
obj.print();
obj.show();
}
}
```

Output:Hello
Welcome

[AA.java](#)

Interfaces: Java 9 feature

In Java 9 and later versions, an interface can have six kinds of things:

1. constant variables
2. **abstract methods**
3. **default methods**
4. static methods
5. private methods
6. private static methods

Example:

```
interface B{  
    void show();  
    default void print(){ // default methods  
        System.out.println("Hello");  
    }  
}
```

```
public class A implements B {  
    public void show(){  
        System.out.println("Hi");  
    }  
    public static void main(String[] args){  
        A a=new A();  
        a.show();  
        a.print();  
    }  
}
```

Interfaces: Java 9 feature

In Java 9 and later versions, an interface can have six kinds of things:

1. constant variables
2. abstract methods
3. default methods
4. **static methods**
5. **private methods**
6. **private static methods**

```
interface BB{  
    void show();  
    default void print(){ // default methods  
        disp();  
        System.out.println("hello");  
    }  
    private void disp(){ //private static or static  
        System.out.println("Private");  
    }  
}  
  
public class AA implements BB {  
    public void show(){  
        System.out.println("hi");  
    }  
    public static void main(String[] args){  
        AA a=new AA();  
        a.show();  
        a.print();  
    }  
}
```

Run time Polymorphism

Runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than at compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Upcasting

When reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



```
class A{}  
class B extends A{}
```

```
A a=new B(); //upcasting
```

Example of Runtime Polymorphism

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, subclass method is invoked at runtime. Since it is determined by the compiler, which method will be invoked at runtime, so it is known as runtime polymorphism.

```
class Bike{  
    void run(){System.out.println("running");}  
}  
  
class Splendor extends Bike{  
    void run(){System.out.println("running safely with 60km");}  
  
    public static void main(String args[]){  
        Bike b = new Splendor();//upcasting  
        b.run();  
    }  
}
```

Output: running safely with 60km.

Runtime Polymorphism with data member

Method is overridden not the datamembers, so runtime polymorphism can't be achieved by data members. In the example given below, both the classes have a datamember speedlimit, we are accessing the datamember by the reference variable of Parent class which refers to the subclass object. Since we are accessing the datamember which is not overridden, hence it will access the datamember of Parent class always.



Rule: Runtime polymorphism can't be achieved by data members.

```
class Bike{  
    int speedlimit=90;  
}  
  
class Honda extends Bike{  
    int speedlimit=150;  
  
    public static void main(String args[]){  
        Bike obj=new Honda();  
        System.out.println(obj.speedlimit);//90  
    }  
}
```

[HondaBike.java](#)

static binding:

When type of the object is determined at compiled time(by the compiler), it is known as static binding.
If there is any private,final or static method in a class,it is static binding.

Example of static binding:

```
class Dog{  
    private void eat(){System.out.println("dog is eating...");}  
  
    public static void main(String args[]){  
        Dog d1=new Dog();  
        d1.eat();  
    }  
}
```

Dynamic binding:

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding:

```
class Animal{
    void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
    void eat(){System.out.println("dog is eating...");}
}

public static void main(String args[]){
    Animal a=new Dog();
    a.eat();
}
```

Output:dog is eating...

In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So compiler doesn't know its type, only its base type.

What is marker or tagged interface ?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

//How Serializable interface is written?

```
public interface Serializable{  
}
```

Interface vs Abstract Class

Abstract Classes

1. Contain one or more abstract methods, can contain no abstract methods also i.e. only concrete methods & can have instance variables.
2. Keyword "abstract"
3. Contain private ,protected and public members.
4. A class **extending** an abstract class need not implement any of the methods defined in the abstract class

Interfaces

1. Contain only method declarations and public static final constants
2. Keyword "interface"
3. Only have public members
4. A class **implementing** an interface must implement all of the methods defined in the interface

Packages

A package is a grouping of related types (class, interface) providing access protection and name space management.

Package

A package is a group of similar types of classes, interfaces and sub-packages.

Package can be categorized in two forms, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

In this page, we will have the detailed learning of creating user-defined packages.

Advantage of Package

- Package is used to categorize the classes and interfaces so that they can be easily maintained.
- Package provides access protection.
- Package removes naming collision.

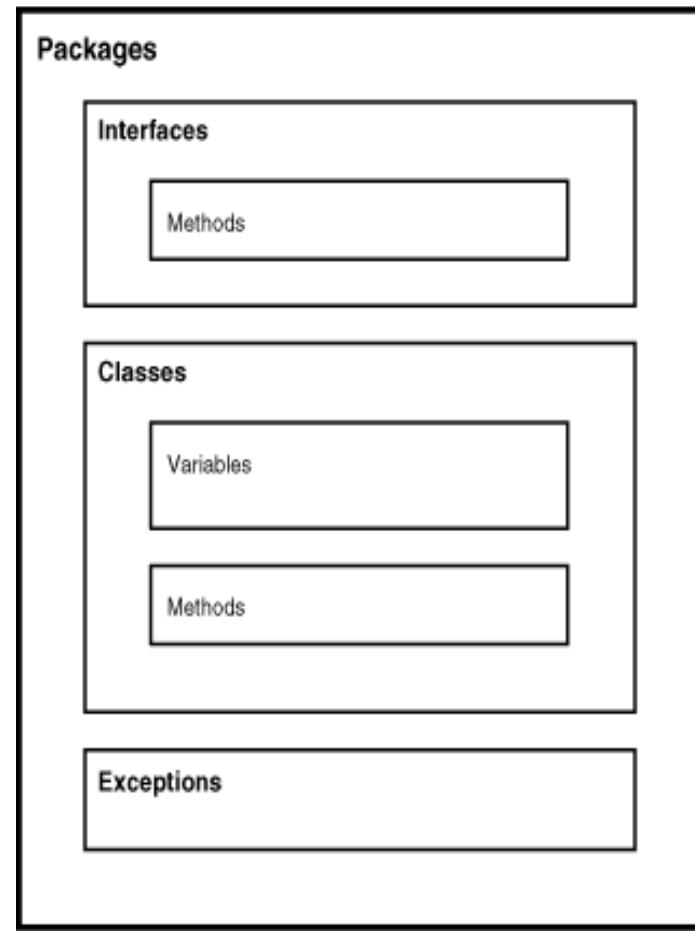
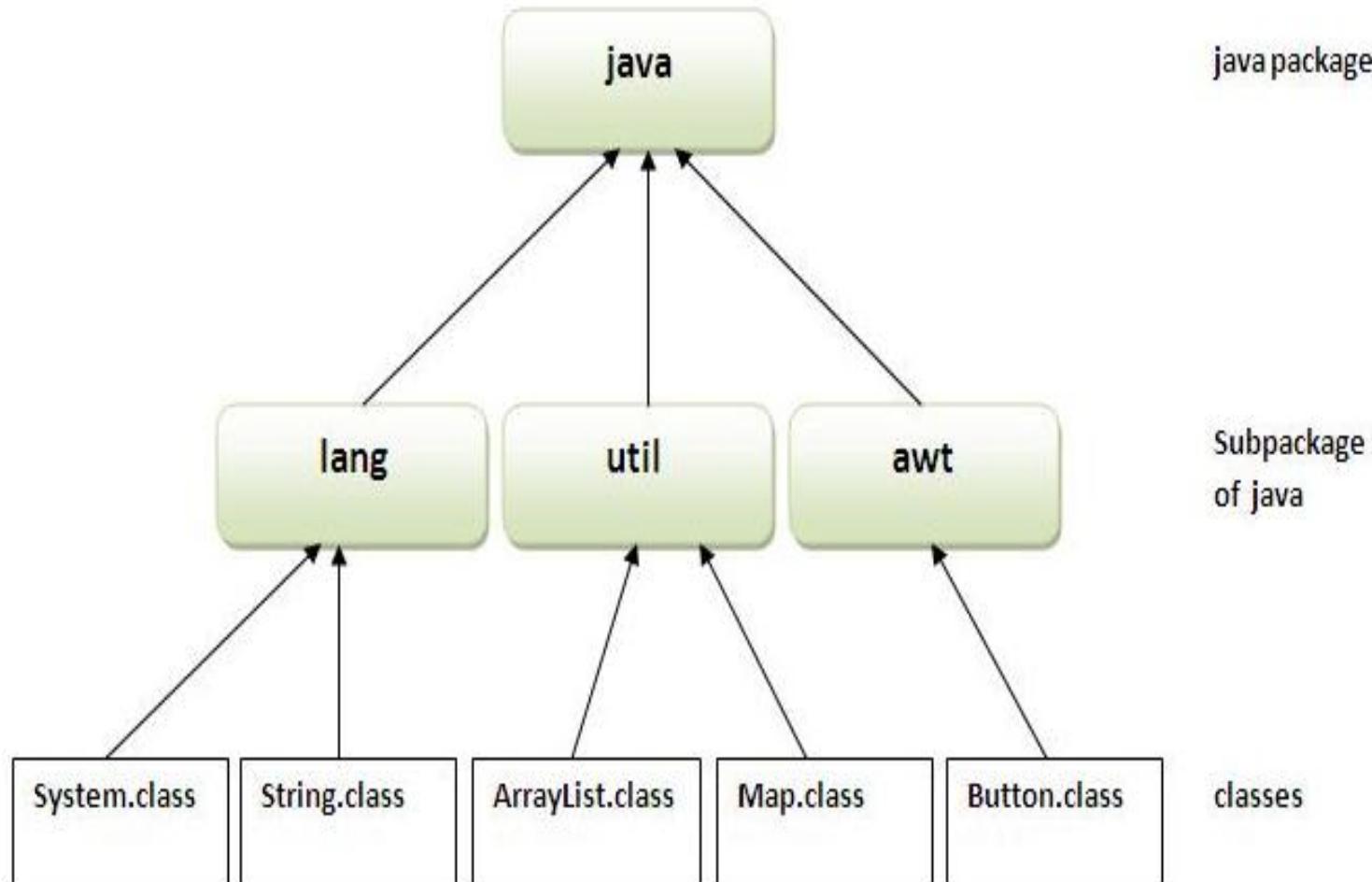


FIGURE 2.10 How Java packages are organized.

Java Package



How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Creating and Using Packages

Defining a package :

```
package package-name;
```

Importing a package :

```
import packagename.classname;
```

or

```
import packagename.*;
```

Naming convention :

```
double y = java.lang.Math.sqrt(x); //fully qualified name
```

Packages – explicit and implicit import

```
import java.util.*; // implicit import
import java.sql.*;
public class Test {

    public static void main(String[] args) {
        Date d=new Date(); // ambiguous ref
        System.out.println(d);
    }
}
```

```
import java.util.Date; // explicit import
import java.sql.*; // implicit import
public class Test {

    public static void main(String[] args) {
        Date d=new Date();
        System.out.println(d);
    }
}
```

static import

```
public class Test {  
    public static void main(String[] args) {  
        System.out.println(Math.max(123, 321)); //321  
        System.out.println(Math.sqrt(784));//28  
        System.out.println(Math.random()); //random  
        System.out.println(Math.pow(8, 3));//512  
    }  
}
```

```
import static java.lang.Math.*; // static import  
public class Test {  
    public static void main(String[] args) {  
        System.out.println(max(123, 321));  
        System.out.println(sqrt(784));  
        System.out.println(random());  
        System.out.println(pow(8, 3));  
    }  
}
```

Simple example of package:

The keyword package is used to create a package.

```
//save as Simple.java

package mypack;
public class Simple{
    public static void main(String args[]){
        System.out.println("Welcome to package");
    }
}
```

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

[Simple1.java](#)

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

*Example of package that import the packagename.**

```
//save by A.java

package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B{
    public static void main(String args[]){
        A obj = new A();
        obj.msg();
    }
}
```

Output:Hello

Example 1 : Sample.java

```
package hyd;
public class Sample{
public void msg () {
System.out.println("Hello i am from hyd package!");
}
public static void main(String args[])
{
    Sample s=new Sample();
    s.msg();
}
}
```

```
//compile // javac -d . Sample.java
// run // java hyd.Sample
```

Example 2 : R.java

```
package cdac;

import hyd.*; // importing package

Public class R {
    public void fun(){
        System.out.println(" Hello, i am from cdac package");
    }

    public static void main(String[] args)
    {
        Sample s1=new Sample();
        s1.msg();
        R r1=new R();
        r1.fun();
    }
}

// javac -d . R.java
// java cdac.R
```

Example 3 :DAC.java

```
package hyd.ameerpet;

public class DAC {
    public void method () {
        System.out.println("Hello, i am from sub package!");
    }
    public static void main(String args[]){
        DAC dac=new DAC();
        dac.method();
    }
}

//compile // javac -d . DAC.java
// run // java hyd.ameerpet.DAC
```

Example 4 : **SR.java**

```
package acts;
import cdac.*;
import hyd.Sample;
class SR{
    public void function(){
        System.out.println(" Hello i am from acts package!");
    }
    public static void main(String[] args) {
        R r=new R(); // from cdac package
        r.fun();
        Sample s=new Sample(); // from hyd package
        s.msg();
        SR sr=new SR(); // from current package-acts
        sr.function();
        hyd.ameerpet.DAC d=new hyd.ameerpet.DAC();
        d.method(); // from sub package
    }
}
```

Note: All classes must be public



If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

Example of package by import fully qualified name

```
//save by A.java

package pack;
public class A{
    public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
class B{
    public static void main(String args[]){
        pack.A obj = new pack.A();//using fully qualified name
        obj.msg();
    }
}
```

Access Modifiers

	private	No modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non- subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Modifiers

There are 12 modifiers in Java :

- public
- private **Visibility modifiers**
- protected
- (default)

- static
- abstract
- final
- strictfp
- native
- synchronized
- transient
- volatile

Modifiers

- The only applicable modifiers for **top level classes** in Java are:
public, default, final, abstract and strictfp
- For the **inner classes**, the following are allowed:
public, default, final, abstract, strictfp, private, protected and static
- **final** – modifier applicable for **classes, methods** and **variables**
- **abstract** - modifier applicable for **classes, methods** but not variables
- **final class** can not have **abstract methods** where as **abstract class** can contain **final methods**
- ```
public class A {
 final int x;
} // variable x is not initialized
```
- ```
public class A {  
    final static int x;  
} // variable x is not initialized
```

Modifiers

strictfp: strict floating point – IEEE 754 standard

strictfp modifier applicable for **classes** and **methods** but not variables

*The only applicable modifier for local variables is final

```
public class A {  
    public void test(){  
        final int x;  
        System.out.println("Hello");  
    }  
}  
// Hello
```

```
public class A {  
    public void test(){  
        final int x;  
        System.out.println(x);  
    }  
}  
// variable x might not have been initialized
```

Modifiers

- **static** modifier – applicable for **variables** and **methods** but not for classes (but **inner classes** can be declared static)
- **native** modifier – applicable for **methods** but not for variables and classes
- synchronized modifier – applicable for **methods** and **blocks** but not for classes and variables
- **transient** modifier - applicable for only **variables**.
- **volatile** modifier – applicable for only **variables**

Modifiers

- The only applicable modifier for local variables is **final**
- The modifiers which are applicable for only variables but not for classes and methods : **volatile** and **transient**
- The modifiers which are applicable for only methods but not for classes and variables : **synchronized** and **native**
- The modifiers which are applicable for top level classes, methods and variables : **public**, **default** and **final**
- The modifiers which are applicable for inner classes but not for outer classes are: **private**, **protected** and **static**
- **final**, **strictfp** and **synchronized** can be used with **main()**
- Ex: **public static final strictfp synchronized void main(String...args)**

modifiers table

modifier	Outer class	Inner class	method	variable	block	interface	enum	constructor	
public	yes	yes	yes	yes	no	yes	yes	yes	
(default)	yes	yes	yes	yes	no	yes	yes	yes	
private	no	yes	yes	yes	no	no	no	yes	
protected	no	yes	yes	yes	no	no	no	yes	
final	yes	yes	yes	yes	no	no	no	no	
abstract	yes	yes	yes	no	no	yes	no	no	
static	no	yes	yes	yes	yes	no	no	no	
synchronized	no	no	yes	no	yes	no	no	no	
native	no	no	yes	no	no	no	no	no	
strictfp	yes	yes	yes	no	no	yes	yes	no	
transient	no	no	no	yes	no	no	no	no	
volatile	no	no	no	yes	no	no	no	no	

Enumeration : enum Keyword

- Enumeration is a list of named constants and these Java enumerations define a class type.
- An *enum type* is a special data type that enables for a variable to be a set of predefined constants
- The variable must be equal to one of the values that have been predefined for it.
- Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY)
- Enumeration is used using the keyword **enum**
- Each item in enum is implicitly declared as **public static final** members

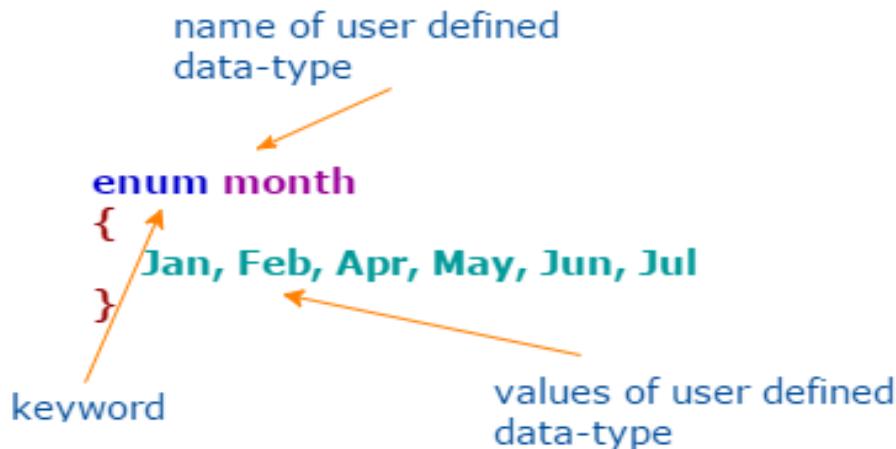


Fig: Java Enumeration

Enumeration : enum Keyword

In the Java programming language, you define an enum type by using the enum keyword. For example, you would specify a days-of-the-week enum type as:

```
public enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY  
}
```

- You should use enum types any time you need to represent a fixed set of constants. That includes natural enum types such as the planets in our solar system and data sets where you know all possible values at compile time
- Java programming language **enum** types are much more powerful than their counterparts in other languages.
- The enum declaration defines a **class** (called an enum type). The enum class body can include **methods and other fields**.
- The compiler automatically adds some special methods when it creates an enum. For example, they have a **static values** method that returns an array containing all of the values of the enum in the order they are declared. This method is commonly used in combination with the for-each construct to iterate over the values of an enum type.
- For example, this code from the Planet class example below iterates over all the planets in the solar system.

```
for (Planet p : Planet.values()) {  
    System.out.printf("Your weight on.....");  
}
```

enum : Example

```
enum Days {  
    MON, TUE, WED, THU, FRI, SAT, SUN;  
}
```

```
public class EnumTest {  
    Days day;  
    public EnumTest(Days day) {  
        this.day = day;  
    }  
    public void daysOfWeek() {  
        switch (day) {  
            case MON:  
                System.out.println("Mondays are bad.");  
                break;  
            case FRI:  
                System.out.println("Fridays are better.");  
                break;  
            case SAT:  
            case SUN:  
                System.out.println("Weekends are best.");  
                break;  
            default:  
                System.out.println("Midweek days are so-so.");  
                break;  
        }  
    }  
}
```

```
public static void main(String[] args) {  
    EnumTest first=new EnumTest(Days.MON);  
    first.daysOfWeek();  
    EnumTest third=new EnumTest(Days.WED);  
    third.daysOfWeek();  
    EnumTest fifth=new EnumTest(Days.FRI);  
    fifth.daysOfWeek();  
    EnumTest sixth=new EnumTest(Days.SAT);  
    sixth.daysOfWeek();  
    EnumTest seven=new EnumTest(Days.SUN);  
    seven.daysOfWeek();  
  
    Days d[]=Days.values();  
    for (Days d1:d)  
        System.out.println(d1);  
    }  
}  
  
/* Mondays are bad.  
Midweek days are so-so.  
Fridays are better.  
Weekends are best.  
Weekends are best  
MON.....  
*/
```

Next

Exception Handling



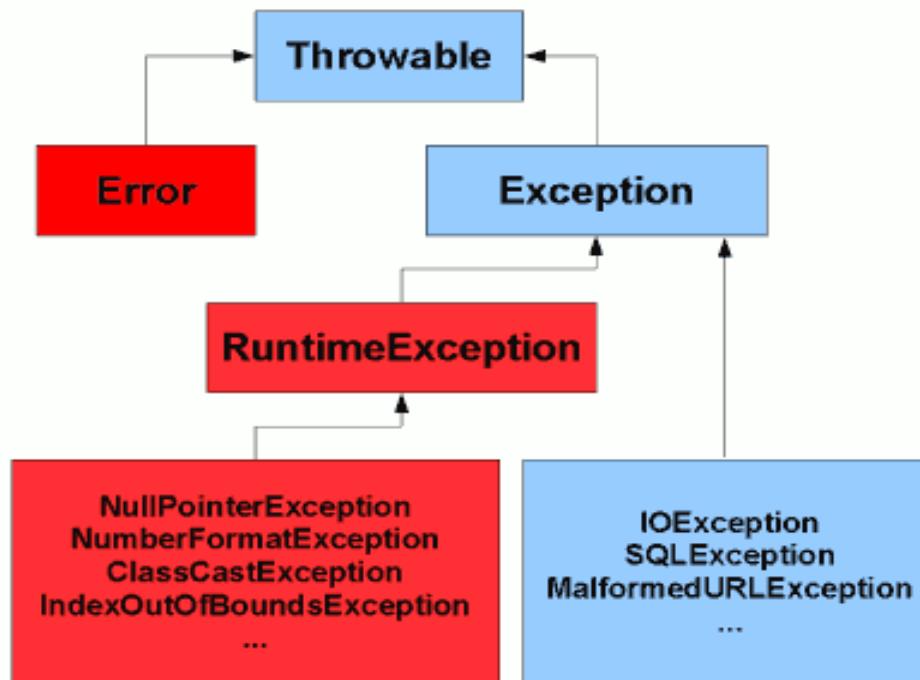
Session 6:

Learning Objectives

- **Describe Exception Handling Mechanism in Java**



Exception Handling



Exception Handling

Kinds of Errors

- Compile time errors
- Run time errors

Source of Errors

- Input Errors
- Device Errors
- Physical Limitations
- Code Errors

What should be done when an error occurred?

- Notify the user if an error occurs
- Save all work
- Allow users to exit from the program smoothly

What is Exception Handling?

- An exception signifies an illegal, invalid or unexpected issue during program execution.
- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions.
- Since exceptions are almost always assumed to be anticipated, you need to provide appropriate exception handling.

Approaches for dealing with error conditions

- Using **Conditional statements** and **return** values
- Use **Java's** exception handling mechanism

division by zero

```
class DivisionByZeroHandled
{
    int c;

    public int compute(int a, int b)
    {
        if(b==0) return 0;
        else
            c=a/b;
        return c;
    }
}
```

Handling Exceptions - Java

Format:

```
try
{
    // Code that may cause an error/exception to occur
}

catch (ExceptionType identifier)
{
    // Code to handle the exception
}
```

Handling Exceptions: DivisionByZero

```
class DivisionByZeroHandled
{
    public static void main(String[] args)
    {
        int a=5, b=0, c=0;
        try
        {
            c=a/b;
        }catch(Exception e){
            System.out.println(e);
        }
        System.out.println("Handled Exception");
    }
}
```

Handling Exceptions: Result Of Calling readLine ()

```
try
{
    System.out.print("Type an integer: ");
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
    String s =br.readLine();The exception can occur here
    System.out.println("You typed in..." + s);
    int num = Integer.parseInt (s);
    System.out.println("Converted to an integer..." + num);
}
```

```
C:\Documents and Settings\sadhu\Desktop\Java Source>javac Ex.java
Ex.java:10: error: unreported exception IOException; must be caught or declared
to be thrown
        String s = br.readLine();^
1 error
```

Handling Exceptions: Result Of Calling parseInt ()

```
try
{
    System.out.print("Type an integer: ");
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
    String s =br.readLine();
    System.out.println("You typed in..." + s);
    num = Integer.parseInt (s);
    System.out.println("Converted to an integer..." + num);
}

```

The second exception
can occur here

```
C:\Documents and Settings\sadhu\Desktop\Java Source>java Ex
Type an integer: You typed in...Hello
Exception in thread "main" java.lang.NumberFormatException: For input string: "Hello"
        at java.lang.NumberFormatException.forInputString(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at java.lang.Integer.parseInt(Unknown Source)
        at Ex.main(Ex.java:12)
```

Where The Exceptions Occur In Class **Integer**?

```
class Integer
{
    public Integer (int value);
    public Integer (String s) throws NumberFormatException;
    public static int parseInt (String s) throws
NumberFormatException;
}
```

Handling Exceptions: Tracing The Example

```
main ()  
try  
{  
    num = Integer.parseInt(s);  
}  
:  
catch (NumberFormatException e)  
{  
    :  
}
```

→ Integer.parseInt (String s)

```
{  
    :  
    :  
}
```

Handling Exceptions: Tracing The Example

```
main ()  
try  
{  
    num = Integer.parseInt (s);  
}  
:  
catch (NumberFormatException e)  
{  
:  
}
```

→ Integer.parseInt (String s)
{
Oops!
The user didn't enter an integer
}

Handling Exceptions: Tracing The Example

```
main ()  
try  
{  
    num = Integer.parseInt (s);  
}  
:  
catch (NumberFormatException e)  
{  
    :  
}
```

```
Integer.parseInt (String s)  
{  
    NumberFormatException e =  
        new NumberFormatException ();  
}
```

Handling Exceptions: Tracing The Example

```
main ()  
try  
{  
    num = Integer.parseInt (s);  
}  
:  
catch (NumberFormatException e)  
{  
    :  
}
```

```
Integer.parseInt (String s)  
{  
    NumberFormatException e =  
        new NumberFormatException ();  
}
```

Handling Exceptions: Tracing The Example

```
main ()  
try  
{  
    num = Integer.parseInt (s);  
}  
:  
catch (NumberFormatException e)  
{  
    Exception must be dealt with here  
}
```

```
Integer.parseInt (String s)
```

```
{
```

```
}
```

Handling Exceptions: Catching The Exception

```
catch (NumberFormatException e)
{
    System.out.println(e);
}
```

Catching The Exception: Error Messages

```
catch (NumberFormatException e)
{
    System.out.println(e.getMessage());
    System.out.println(e);
    e.printStackTrace();
}
```

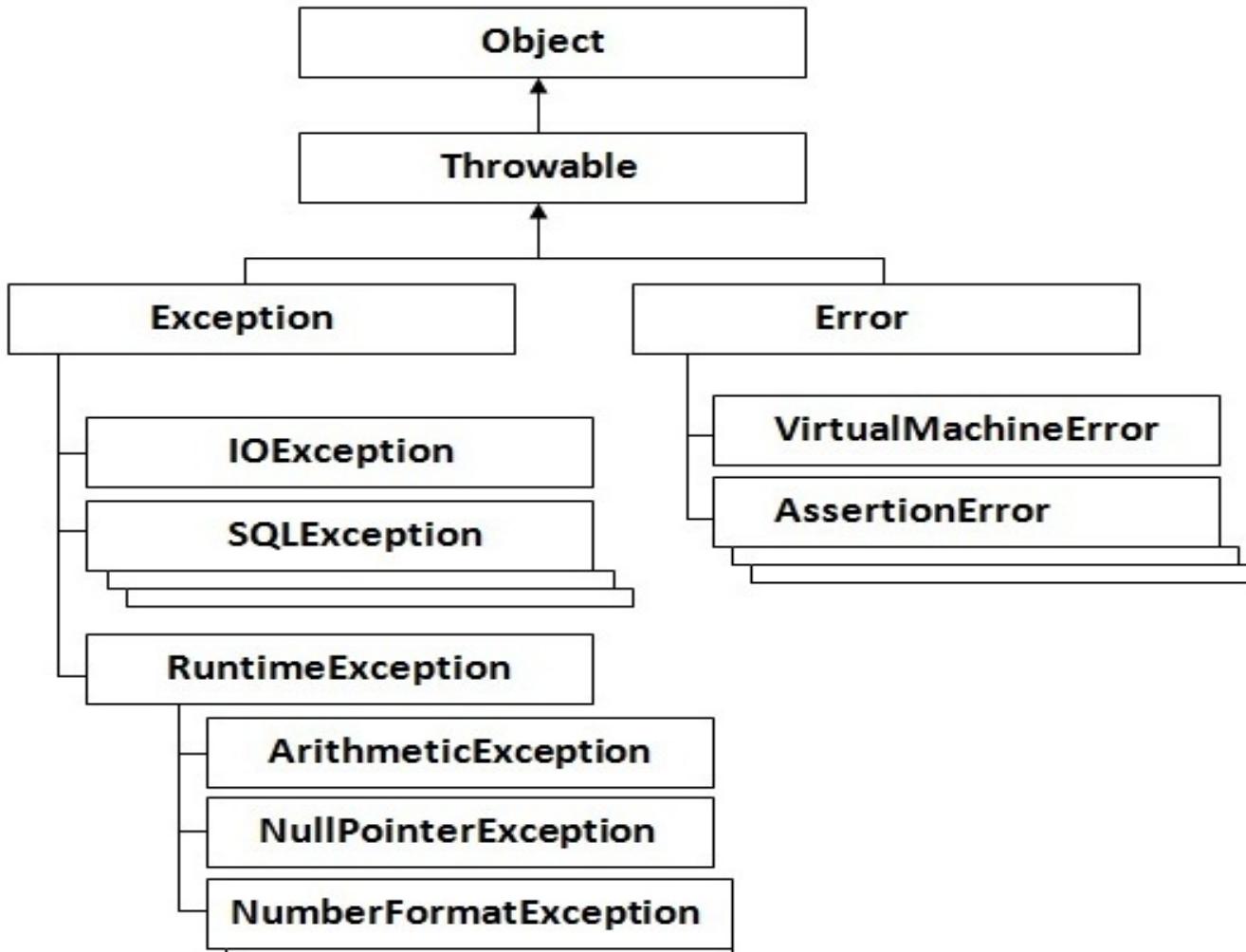
Catching The Exception: Error Messages

```
        catch (NumberFormatException e)
{
    System.out.println(e.getMessage());
    System.out.println(e);
    e.printStackTrace();
}
}
```

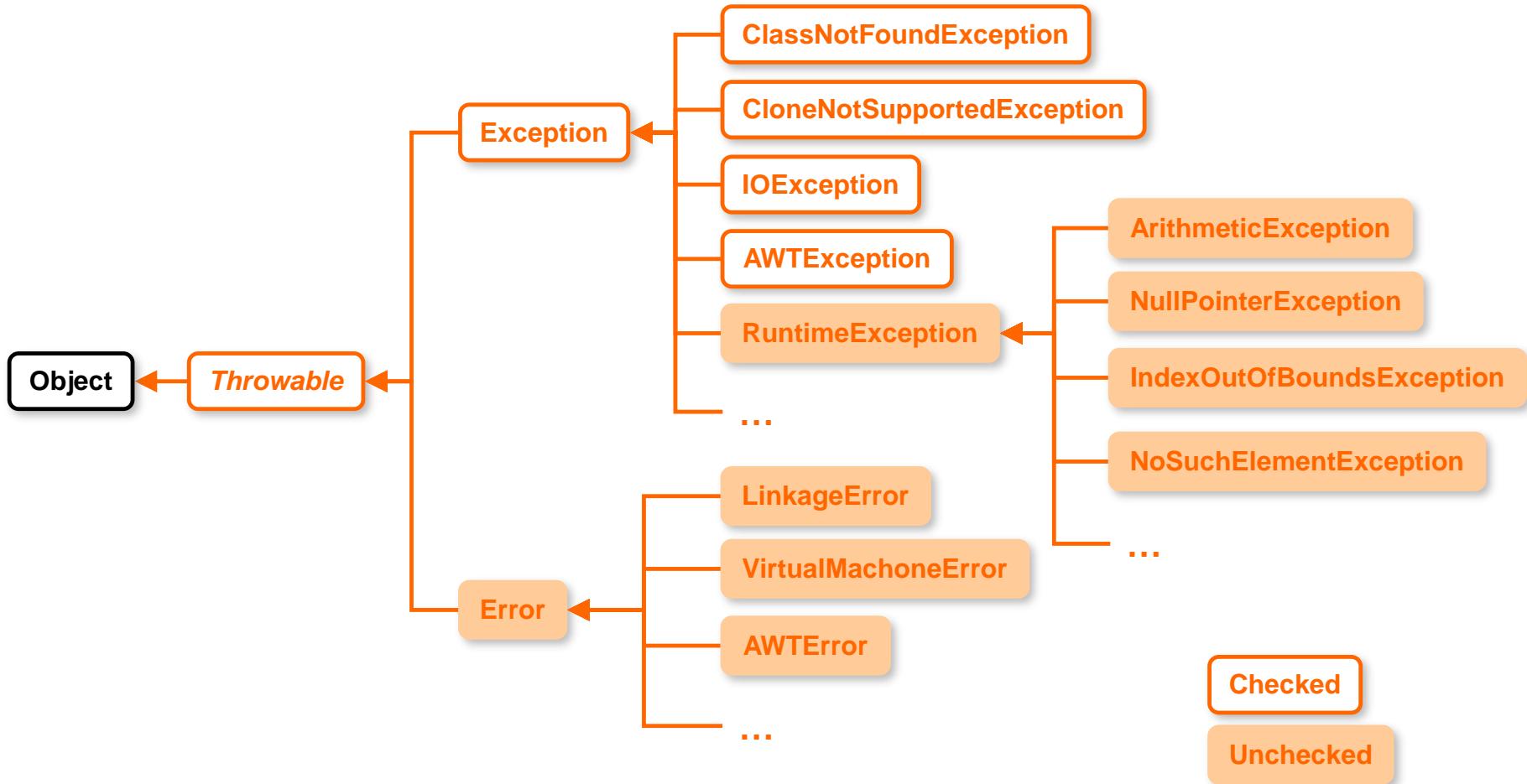
java.lang.NumberFormatException: For input string: "cdac"

at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:426)
at java.lang.Integer.parseInt(Integer.java:476)
at Myclass.main(Myclass.java:39)

Hierarchy of Exception classes



Java Exception class hierarchy



Types of Exception:

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
 2. Unchecked Exception
 3. Error
-

What is the difference between checked and unchecked exceptions ?

1)Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2)Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3)Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionException etc.

Checked Exceptions

- Must be handled if the potential for an error exists
 - must use a **try-catch** block
- Deal with problems that occur in a specific place
 - When a particular method invoked enclose it within a **try-catch** block
- Example:
 - SQLException, IOException

Checked Exceptions

Following is the list of Java Checked Exceptions Defined in `java.lang`.

Exception	Description
<code>ClassNotFoundException</code>	Class not found.
<code>CloneNotSupportedException</code>	Attempt to clone an object that does not implement the <code>Cloneable</code> interface.
<code>IllegalAccessException</code>	Access to a class is denied.
<code>InstantiationException</code>	Attempt to create an object of an abstract class or interface.
<code>InterruptedException</code>	One thread has been interrupted by another thread.
<code>NoSuchFieldException</code>	A requested field does not exist.
<code>NoSuchMethodException</code>	A requested method does not exist.

Characteristics Of Unchecked Exceptions

- The compiler doesn't require you to handle them if they are thrown.
 - *No **try-catch** block required by the **compiler***
- They can occur at any time in the program (not just for a specific method)
- Examples:
 - `NullPointerException`, `IndexOutOfBoundsException`, `ArithmeticException`...

Run Time Exceptions (Unchecked)

Exception	Description
ArithmaticException	Arithmatic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
UnsupportedOperationException	An unsupported operation was encountered.

Common Unchecked Exceptions: NullPointerException

```
int [] arr = null;  
arr[0] = 1;
```

NullPointerException

```
arr = new int [4];  
int i;  
for (i = 0; i <= 4; i++)  
    arr[i] = i;
```

```
arr[i-1] = arr[i-1] / 0;
```

Common Unchecked Exceptions: `ArrayIndexOutOfBoundsException`

```
int [] arr = null;
```

```
arr[0] = 1;
```

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

```
int i;
```

```
for (i = 0; i <= 4; i++)
```

```
    arr[i] = i;
```

```
arr[i-1] = arr[i-1] / 0;
```

ArrayIndexOutOfBoundsException
(when $i = 4$)

Common Unchecked Exceptions: ArithmeticExceptions

```
int [] arr = null;
```

```
arr[0] = 1;
```

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

```
int i;
```

```
for (i = 0; i <= 4; i++)
```

```
    arr[i] = i;
```

```
arr[i-1] = arr[i-1] / 0;
```



ArithmeticException
(Division by zero)

```
int a=50/0;//ArithmaticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;  
System.out.println(s.length());//NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";  
int i=Integer.parseInt(s);//NumberFormatException
```

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[] = new int[5];  
a[10]=50; //ArrayIndexOutOfBoundsException
```

Keywords – Exception Handling in Java

1. try
2. catch
3. finally
4. throw
5. throws

try block

Enclose the code that might throw an exception in try block. It must be used within the method and must be followed by either catch or finally block.

Syntax of try with catch block

```
try{  
...  
}catch(Exception_class_Name reference){}
```

Syntax of try with finally block

```
try{  
...  
}finally{}
```

catch block

Catch block is used to handle the Exception. It must be used after the try block.

Problem without exception handling

```
class Simple{
    public static void main(String args[]){
        int data=50/0;

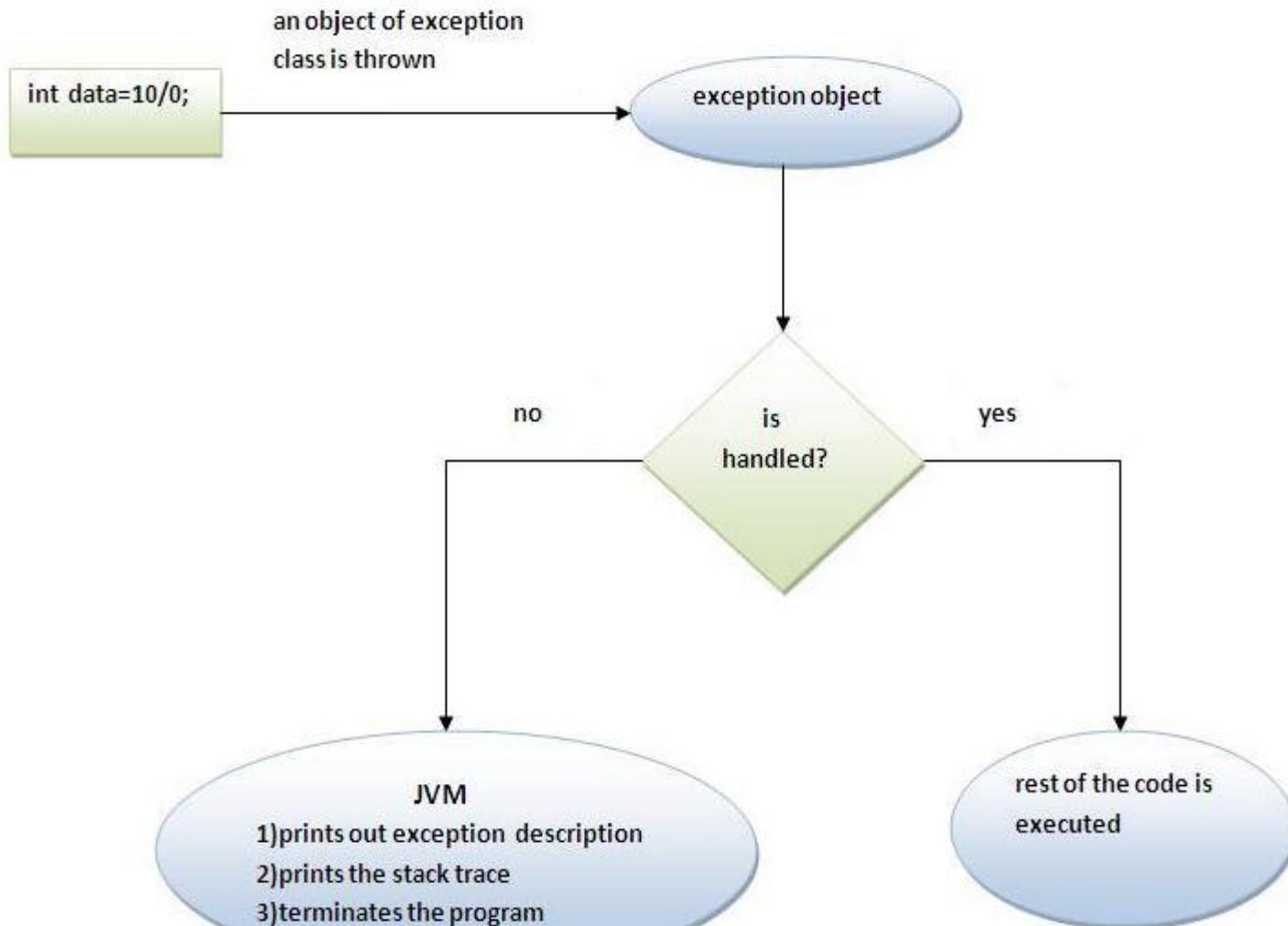
        System.out.println("rest of the code...");
    }
}
```

Output:Exception in thread main java.lang.ArithmetricException:/ by zero

As displayed in the above example, rest of the code is not executed i.e. rest of the code... statement is not printed. Let's see what happens behind the scene:

[SimpleEx.java](#)

What happens behind the code int a=50/0;



Solution by exception handling

```
class Simple{  
    public static void main(String args[]){  
        try{  
            int data=50/0;  
  
        }catch(ArithmaticException e){System.out.println(e);}  
  
        System.out.println("rest of the code...");  
    }  
}
```

```
Output:Exception in thread main java.lang.ArithmaticException:/ by zero  
rest of the code...
```

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

[SimpleEx1.java](#)

Multiple catch block:

If you have to perform different tasks at the occurrence of different Exceptions, use multiple catch block.

<i>Example of multiple **catch block</i>**

```
class Excep4{  
    public static void main(String args[]){  
        try{  
            int a[] = new int[5];  
            a[5] = 30/0;  
        }  
        catch(ArithmaticException e){System.out.println("task1 is completed");}  
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}  
        catch(Exception e){System.out.println("common task completed");}  
  
        System.out.println("rest of the code...");  
    }  
}
```

Output:task1 completed
rest of the code...



Rule:At a time only one Exception is occurred and at a time only one catch block is executed.



Rule:All catch blocks must be ordered from most specific to most general i.e. catch for **ArithmeticException** must come before catch for **Exception** .

```
class Excep4{  
    public static void main(String args[]){  
        try{  
            int a[] = new int[5];  
            a[5] = 30/0;  
        }  
        catch(Exception e){System.out.println("common task completed");}  
        catch(ArithmeticException e){System.out.println("task1 is completed");}  
        catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}  
  
        System.out.println("rest of the code...");  
    }  
}
```

Output:Compile-time error

[Exception2.java](#)

1.7 onwards... multi catch is also provided! Use | symbol

Why use nested try block?

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested

Syntax:

```
....  
try  
{  
    statement 1;  
    statement 2;  
    try  
    {  
        statement 1;  
        statement 2;  
    }  
    catch(Exception e)  
    {  
    }  
}  
catch(Exception e)  
{  
}
```

Example:

```
<b><i>Example of nested try block</i></b>

class Excep6{
    public static void main(String args[]){
        try{
            try{
                System.out.println("going to divide");
                int b =39/0;
            }catch(ArithmeticException e){System.out.println(e);}

            try{
                int a[] =new int[5];
                a[5]=4;
            }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}

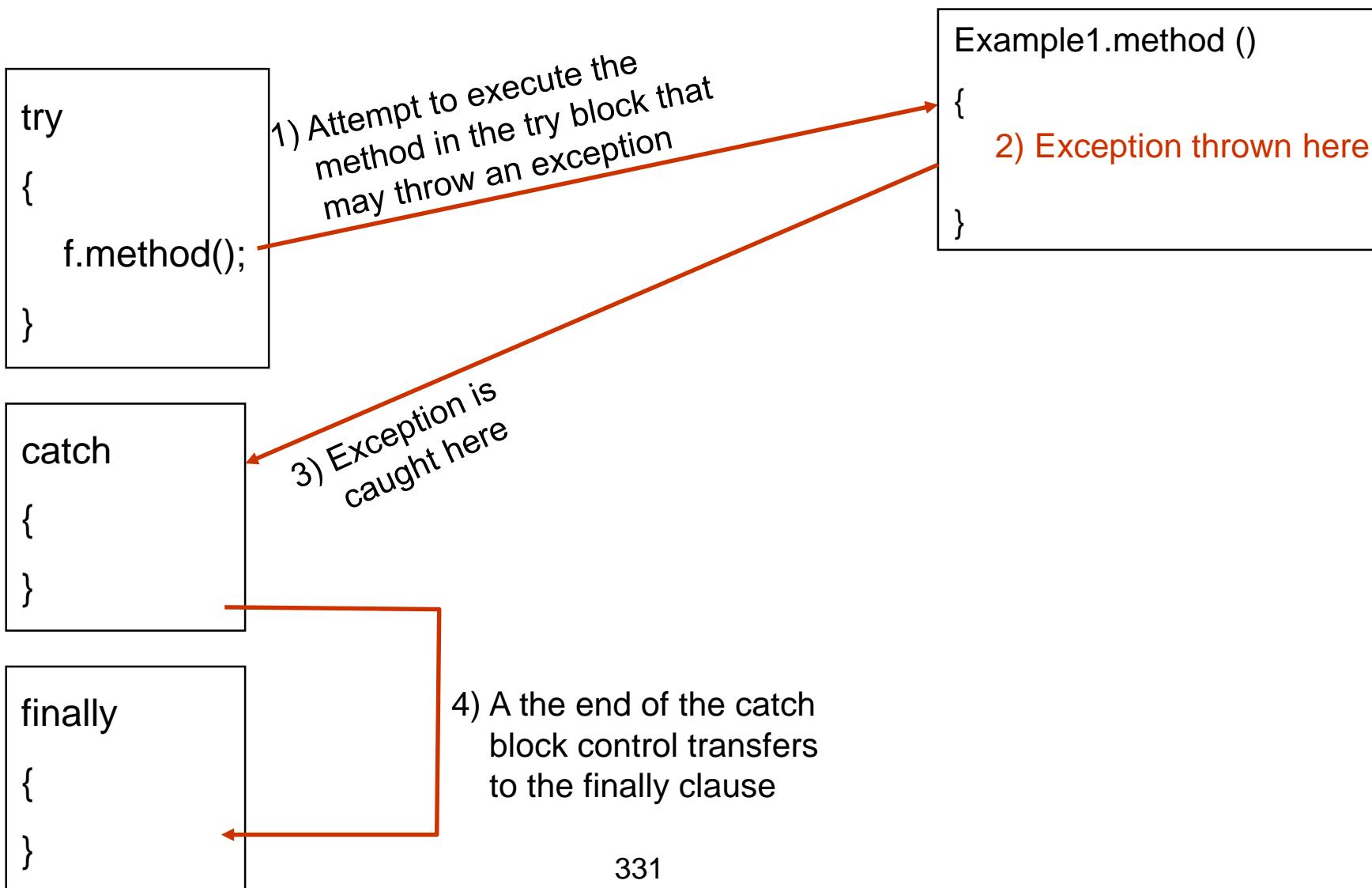
            System.out.println("other statement");
        }catch(Exception e){System.out.println("handled");}
        System.out.println("normal flow..");
    }
}
```

Exception3.java

The Finally Clause

- An additional part of Java's exception handling model (try-catch-finally).
 - Used to enclose statements that must always be executed whether or not an exception occurs.
- ✓ **finally** block will be executed whether or not an exception is thrown.
- ✓ Each try clause requires at least one catch or finally clause.

The Finally Clause: Exception Thrown



The Finally Clause: No Exception Thrown

```
try  
{  
    f.method();  
}
```

1) Attempt to execute the method in the try block that may throw an exception

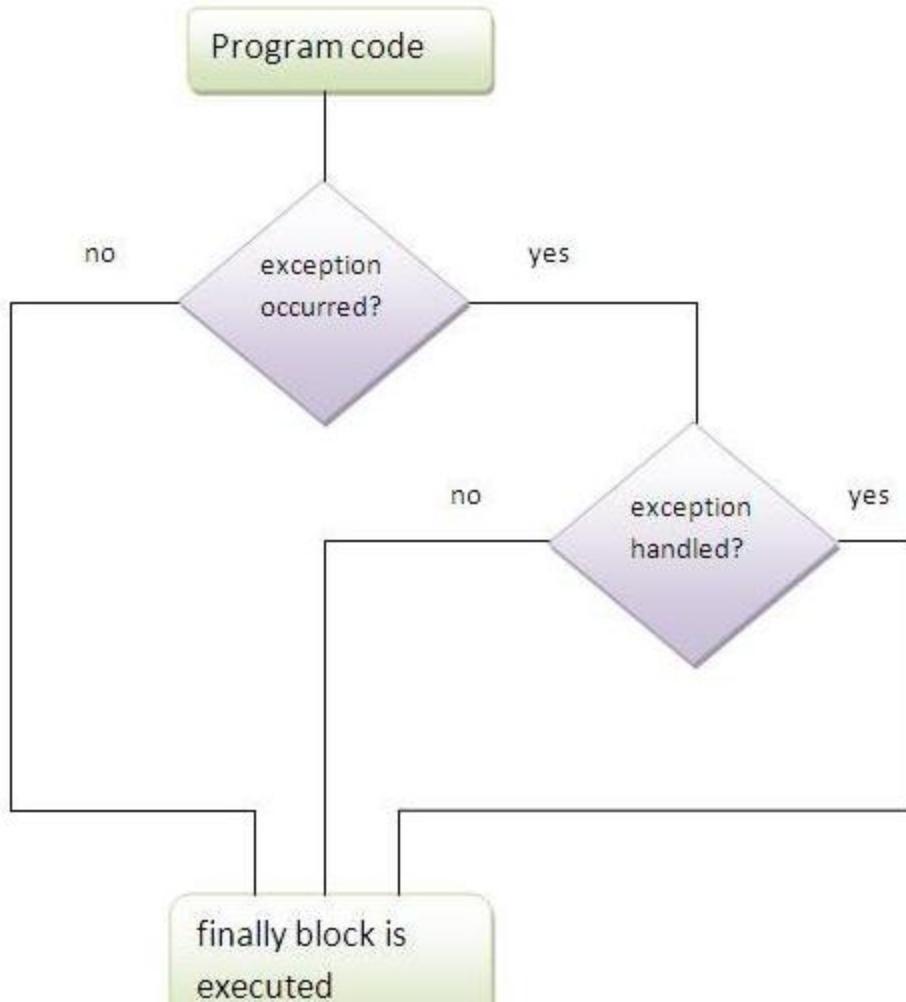
Example1.method ()
{
 2) Code runs okay here
}

```
catch  
{  
}
```

3) At the end of Example.method () control transfers to the finally clause

```
finally  
{  
}
```

The finally block is a block that is always executed. It is mainly used to perform some important tasks such as closing connection, stream etc.



- finally block can be used to put "cleanup" code such as closing a file,closing connection etc.
-

case 1

Program in case exception does not occur

```
class Simple{
    public static void main(String args[]){
        try{
            int data=25/5;
            System.out.println(data);
        }
        catch(NullPointerException e){System.out.println(e);}

        finally{System.out.println("finally block is always executed");}
        System.out.println("rest of the code...");
    }
}
```

Output:

```
5
finally block is always executed
rest of the code...
```

Program in case exception occurred but not handled

```
class Simple{  
    public static void main(String args[]){  
        try{  
            int data=25/0;  
            System.out.println(data);  
        }  
        catch(NullPointerException e){System.out.println(e);}  
  
        finally{System.out.println("finally block is always executed");}  
  
        System.out.println("rest of the code...");  
    }  
}
```

Exception4.java

Output:finally block is always executed
Exception in thread main java.lang.ArithmetricException:/ by zero

throw

- ✓ It is possible for your program to throw an exception explicitly
throw ThrowableInstance
- ✓ Here, *ThrowableInstance* must be an object of type
Throwable or a subclass **Throwable**
- ✓ There are two ways to obtain a **Throwable** objects:
 - ✓ Using a parameter into a catch clause
 - ✓ Creating one with the **new** operator

throw

The throw keyword is used to explicitly throw an exception.

We can throw either checked or unchecked exception. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

Example of throw keyword

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
class Excep13{  
  
    static void validate(int age){  
        if(age<18)  
            throw new ArithmeticException("not valid");  
        else  
            System.out.println("welcome to vote");  
    }  
  
    public static void main(String args[]){  
        validate(13);  
        System.out.println("rest of the code...");  
    }  
}
```

[Exception5.java](#)

Output:Exception in thread main java.lang.ArithmeticException:not valid

Example -throw Statements

```
// Demonstrate throw.
class ThrowDemo {
    static void demoproc() {
        try {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // re-throw the exception
        }
    }

    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

Output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo



Rule: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Program of Exception Propagation

```
class Simple{  
    void m(){  
        int data=50/0;  
    }  
    void n(){  
        m();  
    }  
    void p(){  
        try{  
            n();  
        }catch(Exception e){System.out.println("exception handled");}  
    }  
    public static void main(String args[]){  
        Simple obj=new Simple();  
        obj.p();  
        System.out.println("normal flow...");  
    }  
}
```

Output:exception handled
normal flow...

[Exception6.java](#)



Rule: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Program which describes that checked exceptions are not propagated

```
class Simple{  
    void m(){  
        throw new java.io.IOException("device error");//checked exception  
    }  
    void n(){  
        m();  
    }  
    void p(){  
        try{  
            n();  
        }catch(Exception e){System.out.println("exception handled");}  
    }  
    public static void main(String args[]){  
        Simple obj=new Simple();  
        obj.p();  
        System.out.println("normal flow");  
    }  
}
```

Output:Compile Time Error

Exception7.java

throws keyword:

The throws keyword is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

Syntax of throws keyword:

```
void method_name() throws exception_class_name{  
    ...  
}
```

Que) Which exception should we declare?

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

throws

- ✓ If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception
- ✓ *type method-name parameter-list) throws exception-list*

```
{  
    // body of method  
}
```
- ✓ It is not applicable for **Error** or **RuntimeException**, or any of their subclasses

Example: incorrect program

```
// This program contains an error and will not compile.
class ThrowsDemo {
    static void throwOne() {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        throwOne();
    }
}
```

Example: corrected version

```
// This is now correct.
class ThrowsDemo {
    static void throwOne() throws IllegalAccessException {
        System.out.println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[]) {
        try {
            throwOne();
        } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
        }
    }
}
```

Output:

Inside throwOne.

[ThrowsDemo.java](#)

Caught java.lang.IllegalAccessException: demo

Program which describes that checked exceptions can be propagated by throws keyword.

```
import java.io.IOException;
class Simple{
    void m()throws IOException{
        throw new IOException("device error");//checked exception
    }
    void n()throws IOException{
        m();
    }
    void p(){
        try{
            n();
        }catch(Exception e){System.out.println("exception handled");}
    }
    public static void main(String args[]){
        Simple obj=new Simple();
        obj.p();
        System.out.println("normal flow...");
    }
}
```

Output:exception handled
normal flow...

[Exception8.java](#)

User Defined Exception

- ✓ Define a subclass of the Exception class.
- ✓ The new subclass inherits all the methods of Exception and can override them.

```
class MyException extends Exception{  
    private int a;  
  
    MyException(int i) {  
        a = i;  
    }  
  
    public String toString(){  
        return "MyException [" + a + "]";  
    }  
}
```

Example contd..

```
class test{  
    static void compute (int a) throws Myexception{  
        if(a>10) throw new MyException(a);  
        System.out.println("Normal Exit");  
    }  
    public static void main(String args[]){  
        try{  
            compute(1);  
            compute(20);  
        }catch(MyException e){ System.out.println("Caught " +e);  
    }  
}
```

[test.java](#)

[ExceptionEx.java](#)

```
class MyException extends Exception {  
public MyException (String errorMessage) {  
super (errorMessage);  
}  
}  
  
class MyMarks{  
private int marks=0;  
  
public void setMarks(int mark) throws MyException {  
if(mark>0){  
this.marks=marks;  
}  
else{  
throw new MyException("Invalid Marks");  
}  
}  
  
public int getMarks(){  
return marks;  
}  
}
```

```
public class ExceptionEx  
{  
public static void main(String args[]) {  
try {  
  
MyMarks mm=new MyMarks();  
  
mm.setMarks(-1);  
  
int i=mm.getMarks();  
  
System.out.println(i);  
}  
catch(MyException e)  
{System.out.println(e);}  
}  
}
```

```
public class NilBalanceException extends Exception{  
  
    public NilBalanceException(String errMsg) {  
        super(errMsg);  
    }  
  
}  
  
class Bank{  
    double amount=10000;  
  
    void withdraw(double amt) throws NilBalanceException {  
  
        if(amt>amount){  
            throw new NilBalanceException("Insufficient Funds to  
withdraw!!");  
  
        }  
    }  
}
```

```
public class ExceptionTest {  
  
    public static void main(String[] args) {  
  
        Bank b=new Bank();  
  
        try {  
            b.withdraw(20000);  
        }  
        catch(Exception e){  
            System.out.println(e);  
        }  
    }  
}
```

Difference between throw and throws:

1)throw is used to explicitly throw an exception.	throws is used to declare an exception.
2)checked exception can not be propagated without throw	checked exception can be propagated with throws.
3)throw is followed by an instance.	throws is followed by class.
4)throw is used within the method.	throws is used with the method signature.
5)You cannot throw multiple exception	You can declare multiple exception e.g. public void method()throws IOException,SQLException.

Try with resources

Example1:

```
BR br=null;
try{
    BR br=ner BR();
    br.readLine();// and rest of the code
}
catch(Exception e){
    Handling code
}
finally{
    if(br!=null)
        br.close()
}
```

Example 2: try with resources

```
try(BR br=new BR(new BR(System.in))){
    br.readLine();
}
catch(IOException e){
    Handling code
}

// finally not required

try(r1;r2;r3){
    // resources should be AutoCloseable resources
}
```

Note: 1.7 onwards.... try with resources is possible without catch or finally

try with resources

```
public class TryDemo {  
    public static void main(String[] args) {  
        try(BufferedReader br=new BufferedReader(new InputStreamReader(System.in))){  
            String name= br.readLine();  
            System.out.println("Hi "+name);  
        }catch(IOException e){  
            System.out.println(e);  
        }  
    }  
}  
  
public class TryDemo {  
    public static void main(String[] args) throws IOException {  
        try(BufferedReader br=new BufferedReader(new InputStreamReader(System.in))){  
            String name= br.readLine();  
            System.out.println("Hi "+name);  
        }  
    }  
}  
  
// works well without catch
```

Next

Arrays

String Handling

java.lang

java.util

Session 7:

Learning Objectives

- **Arrays in Java**
- **String Handling**
- **java.lang**
- **java.util**



String Handling

String

Generally string is a sequence of characters. But in java, string is an object. String class is used to create string object.

Do You Know ?

- *Why String objects are immutable ?*
- *How to create an immutable class ?*
- *What is string constant pool ?*
- *What code is written by the compiler if you concat any string by + (string concatenation operator) ?*
- *What is the difference between StringBuffer and StringBuilder class ?*

How to create String object?

There are two ways to create String object:

1. By string literal
 2. By new keyword
-

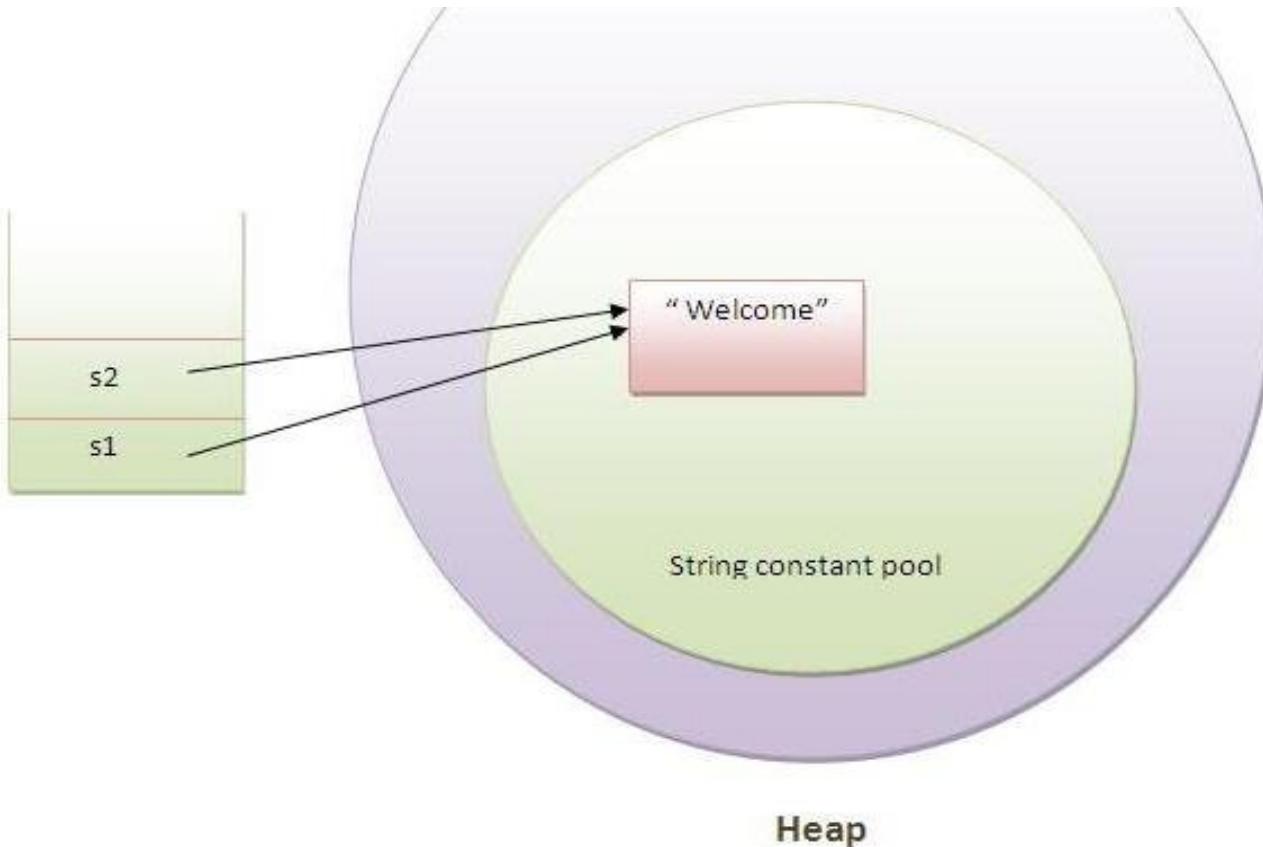
1) String literal:

String literal is created by double quote. For Example:

```
String s="Hello";
```

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object instantiates, then is placed in the pool. For example:

```
String s1="Welcome";
String s2="Welcome";//no new object will be created
```



In the above example only one object will be created. First time JVM will find no string object with the name "Welcome" in string constant pool, so it will create a new object. Second time it will find the string with the name "Welcome" in string constant pool, so it will not create new object whether will return the reference to the same instance.

Note: String objects are stored in a special memory area known as string constant pool inside the Heap memory.

Why java uses concept of string literal?

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

2) By new keyword:

```
String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, JVM will create a new String object in normal(nonpool) Heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in Heap(nonpool).

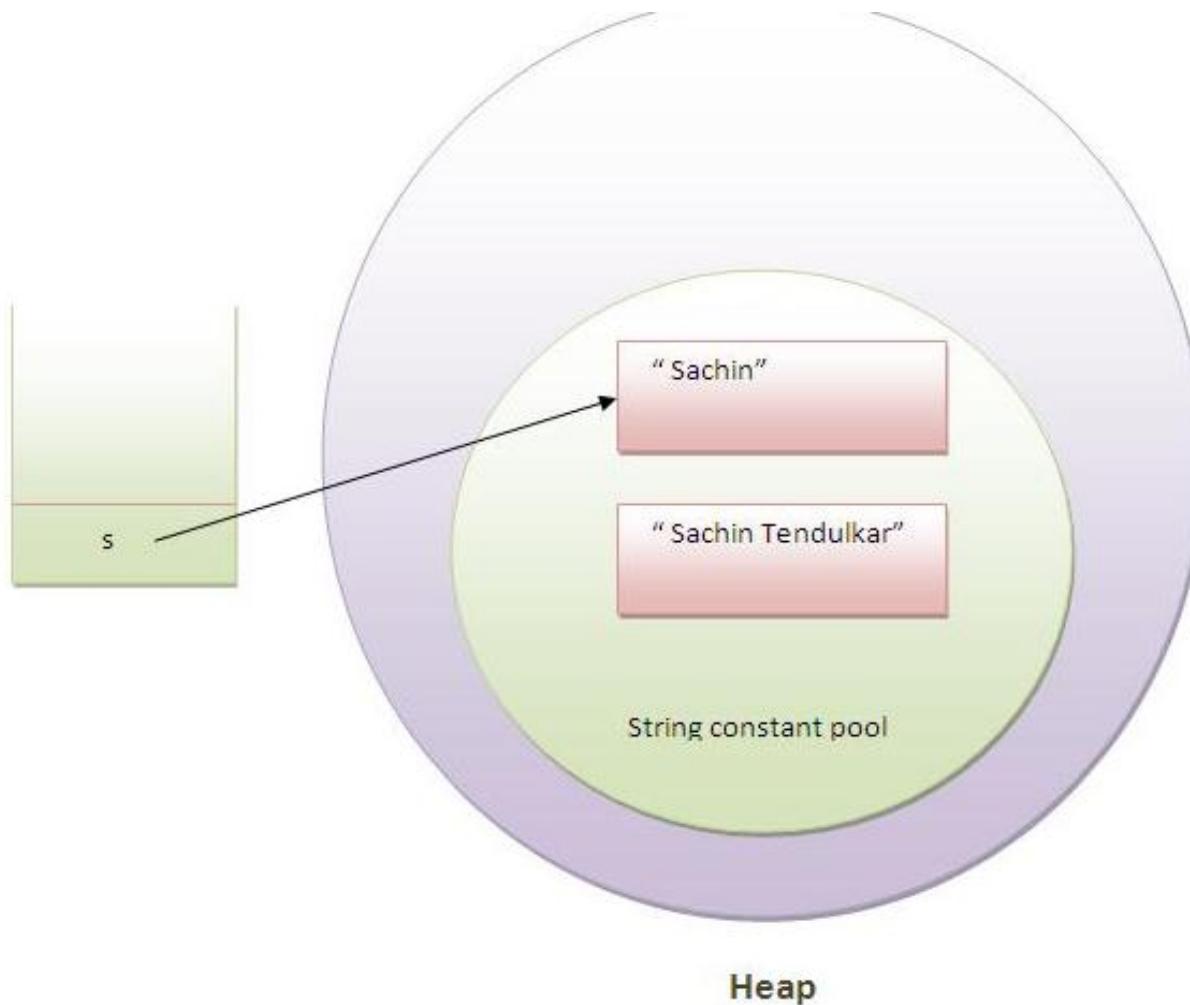
Immutable String:

In java, strings are immutable (unmodifiable) objects. For example

```
class Simple{  
    public static void main(String args[]){  
  
        String s="Sachin";  
        s.concat(" Tendulkar");//concat() method appends the string at the end  
        System.out.println(s);//will print Sachin because strings are immutable objects  
    }  
}
```

Output: Sachin

[S1.java](#)



As you can see in the above figure that two objects will be created but no reference variable refers to "Sachin Tendulkar". But if we explicitly assign it to the reference variable, it will refer to "Sachin Tendulkar"

String comparison in Java

We can compare two given on the basis of content and reference. It is used in authentication (`equals()` method), sorting (`compareTo()` method) etc.

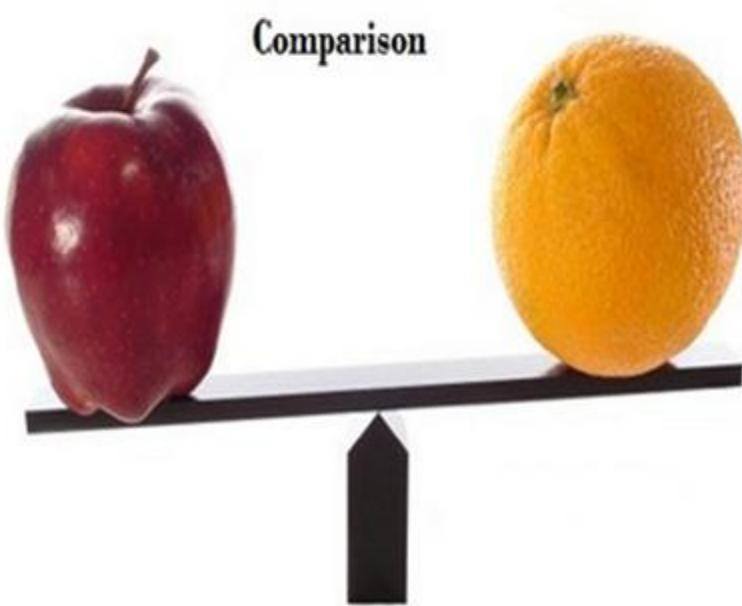
There are three ways to compare String objects:

1. By `equals()` method
2. By `==` operator
3. By `compareTo()` method

1) By `equals()` method:

`equals()` method compares the original content of the string. It compares values of string for equality. String class provides two methods:

- **`public boolean equals(Object another){}`** compares this string to the specified object.
- **`public boolean equalsIgnoreCase(String another){}`** compares this String to another String, ignoring case.



```
//<b><i>Example of equals(Object) method</i></b>

class Simple{
    public static void main(String args[]){
        String s1="Sachin";
        String s2="Sachin";
        String s3=new String("Sachin");
        String s4="Saurav";

        System.out.println(s1.equals(s2));//true
        System.out.println(s1.equals(s3));//true
        System.out.println(s1.equals(s4));//false
    }
}
```

```
Output:true
      true
      false
```

```
//<b><i>Example of equalsIgnoreCase(String) method</i></b>

class Simple{
    public static void main(String args[]){

        String s1="Sachin";
        String s2="SACHIN";

        System.out.println(s1.equals(s2));//false
        System.out.println(s1.equalsIgnoreCase(s2));//true
    }
}
```

Output: false
true

2) By == operator:

The == operator compares references not values.

```
//<b><i>Example of == operator</i></b>

class Simple{
    public static void main(String args[]){
        String s1="Sachin";
        String s2="Sachin";
        String s3=new String("Sachin");

        System.out.println(s1==s2);//true (because both refer to same instance)
        System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
    }
}
```

Output:
true
false

3) By compareTo() method:

compareTo() method compares values and returns an int which tells if the values compare less than, equal, or greater than.

Suppose s1 and s2 are two string variables.If:

- **s1 == s2** :0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

```
//<b><i>Example of compareTo() method:</i></b>

class Simple{
    public static void main(String args[]){
        String s1="Sachin";
        String s2="Sachin";
        String s3="Ratan";

        System.out.println(s1.compareTo(s2));//0
        System.out.println(s1.compareTo(s3));//1(because s1>s3)
        System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
    }
}
```

String Concatenation in Java

Concatenating strings form a new string i.e. the combination of multiple strings.

There are two ways to concat string objects:

1. By + (string concatenation) operator
2. By concat() method

1) By + (string concatenation) operator

String concatenation operator is used to add strings. For Example:

```
//Example of string concatenation operator

class Simple{
    public static void main(String args[]){
        String s="Sachin"+" Tendulkar";
        System.out.println(s); //Sachin Tendulkar
    }
}
```

Output: Sachin Tendulkar

The compiler transforms this to:

```
String s=(new StringBuilder()).append("Sachin").append(" Tendulkar).toString();
```

String concatenation is implemented through the `StringBuilder`(or `StringBuffer`) class and its `append` method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For Example:

```
class Simple{  
    public static void main(String args[]){  
  
        String s=50+30+"Sachin"+40+40;  
        System.out.println(s); //80Sachin4040  
    }  
}
```

Output:80Sachin4040

Note: If either operand is a string, the resulting operation will be string concatenation. If both operands are numbers, the operator will perform an addition.

2) By concat() method

concat() method concatenates the specified string to the end of current string.

Syntax: public String concat(String another){}

```
//<b><i>Example of concat(String) method</i></b>

class Simple{
    public static void main(String args[]){

        String s1="Sachin ";
        String s2="Tendulkar";

        String s3=s1.concat(s2);

        System.out.println(s3); //Sachin Tendulkar
    }
}
```

Output: Sachin Tendulkar

Substring in Java

A part of string is called **substring**. In other words, substring is a subset of another string.

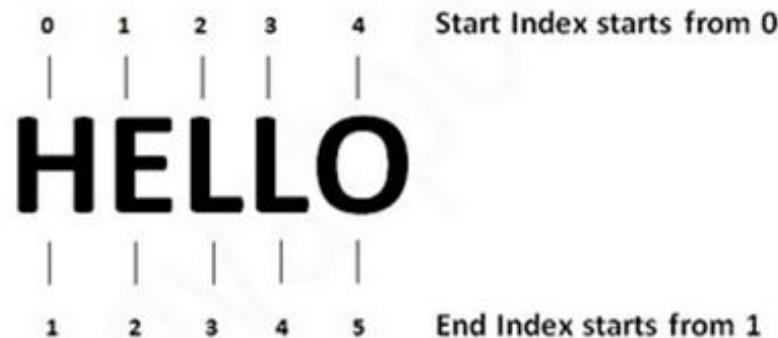
In case of substring startIndex starts from 0 and endIndex starts from 1 or startIndex is inclusive and endIndex is exclusive.

You can get substring from the given String object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex,int endIndex):** This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

- **startIndex:** starts from index 0(inclusive).
- **endIndex:** starts from index 1(exclusive).



Example of java substring

```
//Example of substring() method

class Simple{
    public static void main(String args[]){
        String s="Sachin Tendulkar";
        System.out.println(s.substring(6));//Tendulkar
        System.out.println(s.substring(0,6));//Sachin
    }
}
```

Output:
Tendulkar
Sachin

Methods of String class

java.lang.String class provides a lot of methods to work on string. Let's see the commonly used methods of String class.

Method	Description
1)public boolean equals(Object anObject)	Compares this string to the specified object.
2)public boolean equalsIgnoreCase(String another)	Compares this String to another String, ignoring case.
3)public String concat(String str)	Concatenates the specified string to the end of this string.
4)public int compareTo(String str)	Compares two strings and returns int
5)public int compareToIgnoreCase(String str)	Compares two strings, ignoring case differences.
6)public String substring(int beginIndex)	Returns a new string that is a substring of this string.
7)public String substring(int beginIndex,int endIndex)	Returns a new string that is a substring of this string.
8)public String toUpperCase()	Converts all of the characters in this String to upper case
9)public String toLowerCase()	Converts all of the characters in this String to lower

10)public String trim()	Returns a copy of the string, with leading and trailing whitespace omitted.
11)public boolean startsWith(String prefix)	Tests if this string starts with the specified prefix.
12)public boolean endsWith(String suffix)	Tests if this string ends with the specified suffix.
13)public char charAt(int index)	Returns the char value at the specified index.
14)public int length()	Returns the length of this string.
15)public String intern()	Returns a canonical representation for the string object.

First seven methods have already been discussed. Now Let's take the example of other methods:

toUpperCase() and toLowerCase() method

```
//<b><i>Example of toUpperCase() and toLowerCase() method</i></b>

class Simple{
    public static void main(String args[]){
        String s="Sachin";
        System.out.println(s.toUpperCase());//SACHIN
        System.out.println(s.toLowerCase());//sachin
    }
}
```

```
class Simple{
    public static void main(String args[]){
        String s="Sachin";
        System.out.println(s.startsWith("Sa"));//true
        System.out.println(s.startsWith("n"));//true
    }
}
```

Output: true
true

charAt() method

```
//<b><i>Example of charAt() method</i></b>

class Simple{
    public static void main(String args[]){
        String s="Sachin";
        System.out.println(s.charAt(0));//S
        System.out.println(s.charAt(3));//h
    }
}
```



StringBuffer class:

The StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class is same as String except it is mutable i.e. it can be changed.

Note: StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously . So it is safe and will result in an order.

Commonly used Constructors of StringBuffer class:

1. **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
 2. **StringBuffer(String str):** creates a string buffer with the specified string.
 3. **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.
-

Commonly used methods of StringBuffer class:

1. **public synchronized StringBuffer append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2. **public synchronized StringBuffer insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3. **public synchronized StringBuffer replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.
4. **public synchronized StringBuffer delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.
5. **public synchronized StringBuffer reverse():** is used to reverse the string.
6. **public int capacity():** is used to return the current capacity.
7. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
8. **public char charAt(int index):** is used to return the character at the specified position.
9. **public int length():** is used to return the length of the string i.e. total number of characters.
10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.

What is mutable string?

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

simple example of StringBuffer class by append() method

The append() method concatenates the given argument with this string.

```
class A{  
    public static void main(String args[]){  
  
        StringBuffer sb=new StringBuffer("Hello ");  
        sb.append("Java");//now original string is changed  
  
        System.out.println(sb);//prints Hello Java  
    }  
}
```

```
class A{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello");
    sb.replace(1,3,"Java");
    System.out.println(sb); //prints HJava
}
}
```

Example of delete() method of StringBuffer class

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
class A{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer("Hello");
    sb.delete(1,3);
    System.out.println(sb); //prints Hlo
}
}
```

Example of capacity() method of StringBuffer class

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by $(\text{oldcapacity} * 2) + 2$. For example if your current capacity is 16, it will be $(16 * 2) + 2 = 34$.

```
class A{
public static void main(String args[]){
    StringBuffer sb=new StringBuffer();
    System.out.println(sb.capacity());//default 16

    sb.append("Hello");
    System.out.println(sb.capacity());//now 16

    sb.append("java is my favourite language");
    System.out.println(sb.capacity());//now  $(16 * 2) + 2 = 34$  i.e  $(\text{oldcapacity} * 2) + 2$ 
}
}
```

StringBuilder class:

The `StringBuilder` class is used to create mutable (modifiable) string. The `StringBuilder` class is same as `StringBuffer` class except that it is non-synchronized. It is available since JDK1.5.

Commonly used Constructors of `StringBuilder` class:

1. **`StringBuilder()`**: creates an empty string Builder with the initial capacity of 16.
2. **`StringBuilder(String str)`**: creates a string Builder with the specified string.
3. **`StringBuilder(int length)`**: creates an empty string Builder with the specified capacity as length.

Commonly used methods of StringBuilder class:

1. **public StringBuilder append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2. **public StringBuilder insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3. **public StringBuilder replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.
4. **public StringBuilder delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.
5. **public StringBuilder reverse():** is used to reverse the string.
6. **public int capacity():** is used to return the current capacity.
7. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
8. **public char charAt(int index):** is used to return the character at the specified position.
9. **public int length():** is used to return the length of the string i.e. total number of characters.
10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.

The append() method concatenates the given argument with this string.

```
class A{
public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello ");
sb.append("Java");//now original string is changed

System.out.println(sb);//prints Hello Java
}
}
```

Example of insert() method of StringBuilder class

The insert() method inserts the given string with this string at the given position.

```
class A{
public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello ");
sb.insert(1,"Java");//now original string is changed

System.out.println(sb);//prints HJavaello
}
```

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
class A{
public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello");
sb.replace(1,3,"Java");

System.out.println(sb); //prints HJava
}
}
```

Example of delete() method of StringBuilder class

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

```
class A{
public static void main(String args[]){

StringBuilder sb=new StringBuilder("Hello");
sb.delete(1,3);

System.out.println(sb); //prints Hlo
}
}
```

How to create Immutable class?

TOP PREV NEXT

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members as the example given below:

Example to create Immutable class

In this example, we have created a final class named Employee. It have one final datamember, a parameterized constructor and getter method.

```
public final class Employee{  
    final String pancardNumber;  
  
    public Employee(String pancardNumber){  
        this.pancardNumber=pancardNumber;  
    }  
  
    public String getPancardNumber(){  
        return pancardNumber;  
    }  
}
```

The above class is immutable because:

- The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
- The class is final so we cannot create the subclass.
- There is no setter methods i.e. we have no option to change the value of the instance variable.

Object class

Understanding `toString()` method

[`<<prev`](#)

If you want to represent any object as a string, **`toString()` method** comes into existence.

The `toString()` method returns the string representation of the object.

If you print any object, java compiler internally invokes the `toString()` method on the object. So overriding the `toString()` method, returns the desired output, it can be the state of an object etc. depends on your implementation.

Advantage of the `toString()` method

By overriding the `toString()` method of the Object class, we can return values of the object, so we don't need to write much code.

Let's see the simple code that prints reference.

```
class Student{
    int rollno;
    String name;
    String city;

    Student(int rollno, String name, String city){
        this.rollno=rollno;
        this.name=name;
        this.city=city;
    }

    public static void main(String args[]){
        Student s1=new Student(101,"Raj","lucknow");
        Student s2=new Student(102,"Vijay","ghaziabad");

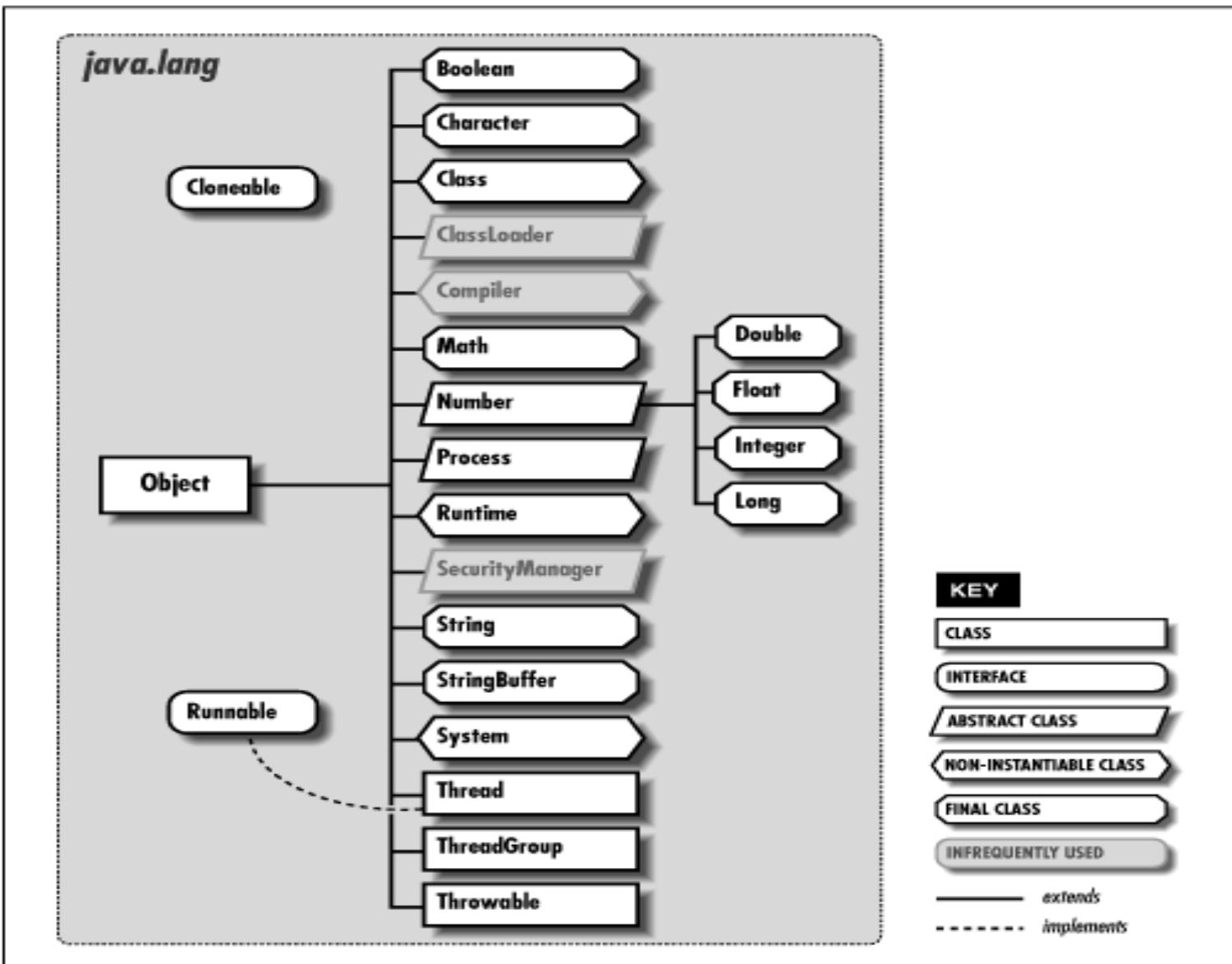
        System.out.println(s1);//compiler writes here s1.toString()
        System.out.println(s2);//compiler writes here s2.toString()
    }
}
```

Output: Student@1fee6fc
Student@1eed786

Now let's see the real example of `toString()` method.

```
class Student{  
    int rollno;  
    String name;  
    String city;  
  
    Student(int rollno, String name, String city){  
        this.rollno=rollno;  
        this.name=name;  
        this.city=city;  
    }  
  
    public String toString(){//overriding the toString() method  
        return rollno+" "+name+" "+city;  
    }  
    public static void main(String args[]){  
        Student s1=new Student(101,"Raj","lucknow");  
        Student s2=new Student(102,"Vijay","ghaziabad");  
  
        System.out.println(s1);//compiler writes here s1.toString()  
        System.out.println(s2);//compiler writes here s2.toString()  
    }  
}
```

java.lang package



java.lang package

The package `java.lang` contains classes and interfaces that are essential to the Java language. These include:

- `Object`, the ultimate superclass of all classes in Java
- `Thread`, the class that controls each thread in a multithreaded program
- `Throwable`, the superclass of all error and exception classes in Java
- Classes that encapsulate the primitive data types in Java
- Classes for accessing system resources and other low-level entities
- `Math`, a class that provides standard mathematical methods
- `String`, the class that is used to represent strings

java.lang.Number

- The **java.lang.Number** class is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short
- The Subclasses of Number must provide methods to convert the represented numeric value to byte, double, float, int, long, and short

Declaration for java.lang.Number class:

```
public abstract class Number extends Object implements Serializable
```

Constructor:

- **Number()** -This is the Single Constructor

Methods:

- [byte byteValue\(\)](#) -This method returns the value of the specified number as a byte
- [abstract double doubleValue\(\)](#) -This method returns the value of the specified number as a double
- [abstract float floatValue\(\)](#) - This method returns the value of the specified number as a float
- [abstract int intValue\(\)](#) - This method returns the value of the specified number as a int
- [abstract long longValue\(\)](#) - This method returns the value of the specified number as a long.
- [short shortValue\(\)](#) - This method returns the value of the specified number as a short

java.lang.Math

- The **java.lang.Math** class contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
- **Declaration for java.lang.Math class:**
public final class Math extends Object
- **Fields:** static double E and static double PI

Methods:

static double abs(double a) - This method returns the absolute value of a double value

static double acos(double a) - This method returns the arc cosine of a value

static double ceil(double a) - Returns the smallest double value that is \geq to the argument

static double cos(double a) - This method returns the trigonometric cosine of an angle

static double exp(double a) - Returns Euler's number e raised to the power of a double value

static double floor(double a) Returns the largest double value that is \leq to the argument

static double log(double a) - Returns the natural logarithm (base e) of a double value

static double max(double a, double b) - Returns the greater of two double values

static double min(double a, double b) - Returns the smaller of two double values

static double pow(double a, double b) - Returns the value of the first argument raised to the power of the second argument.

static double random() - Returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

static long round(double a) - This method returns the closest long to the argument

static double sqrt(double a) - Returns the correctly rounded positive square root of a double value.

[java.lang.System](#)

The **java.lang.System** class contains several useful class fields and methods.

It cannot be instantiated.

Facilities provided by System:

- standard output
- error output streams
- standard input and access to externally defined properties and environment variables.
- A utility method for quickly copying a portion of an array.
- a means of loading files and libraries

Fields:

- **static PrintStream err** -- This is the "standard" error output stream
- **static InputStream in** -- This is the "standard" input stream
- **static PrintStream out** -- This is the "standard" output stream

Methods:

[static void arraycopy\(Object src, int srcPos, Object dest, int destPos, int length\)](#) -

It copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array

[static Console console\(\)](#) -

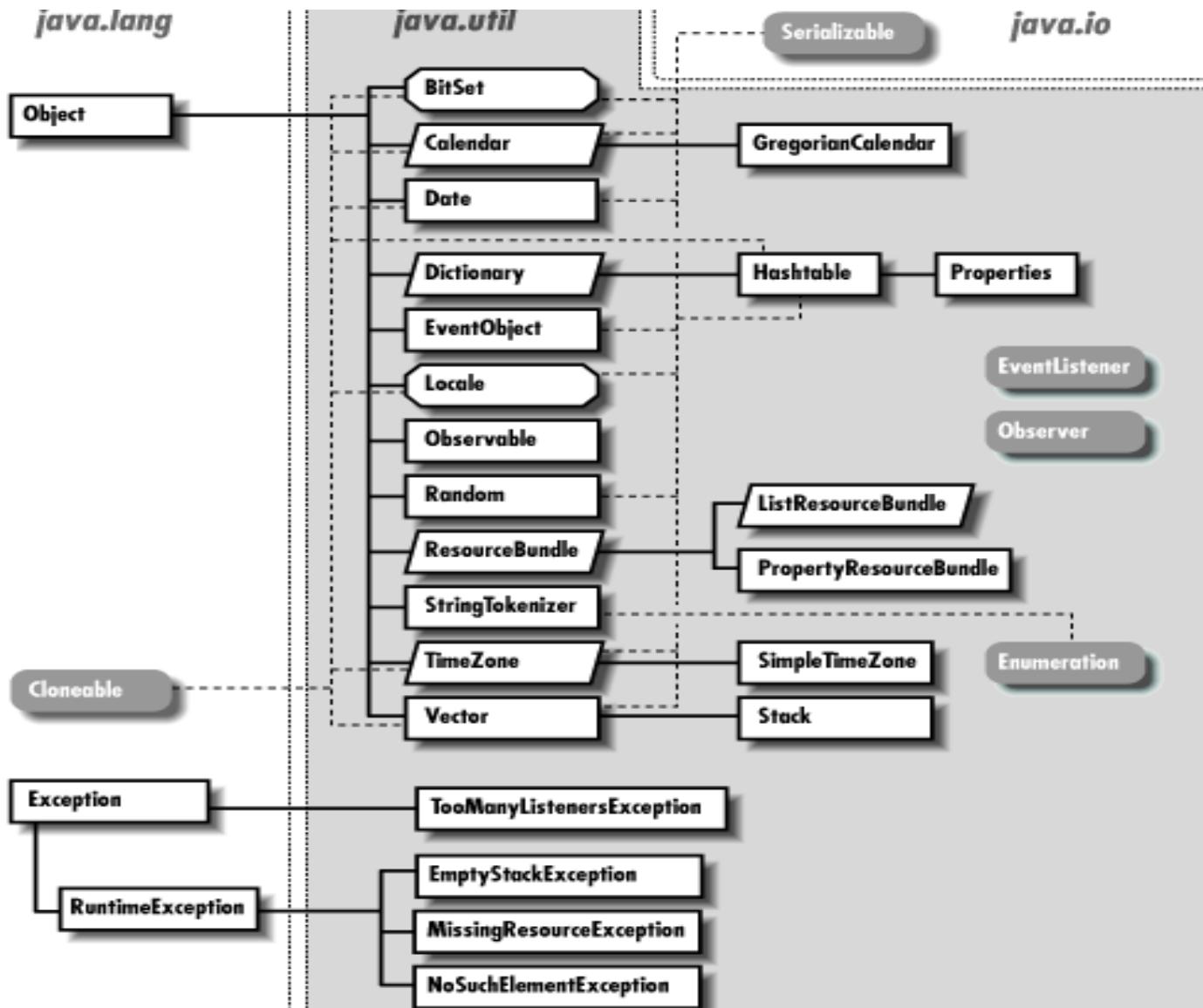
Returns the unique Console object associated with the current Java virtual machine, if any

[static void gc\(\)](#) - This method runs the garbage collector

[static Properties getProperties\(\)](#) - Determines the current system properties

[static Console console\(\)](#) - Returns the unique Console object associated with the current Java virtual machine, if any.

java.util package



java.util package – working with Date and Scanner

- The **java.util.Date** class represents a specific instant in time, with millisecond precision
- The **java.util.Scanner** class is a simple text scanner which can parse primitive types and strings using regular expression

Example:

```
import java.util.*;  
public class DateDemo {  
    public static void main(String[] args) {  
        Date d=new Date();  
        System.out.println(d);  
  
        Scanner sc=new Scanner(System.in);  
        System.out.println("Enetr Your Name:");  
        String name=sc.next();  
        System.out.println(name);  
    }  
}
```

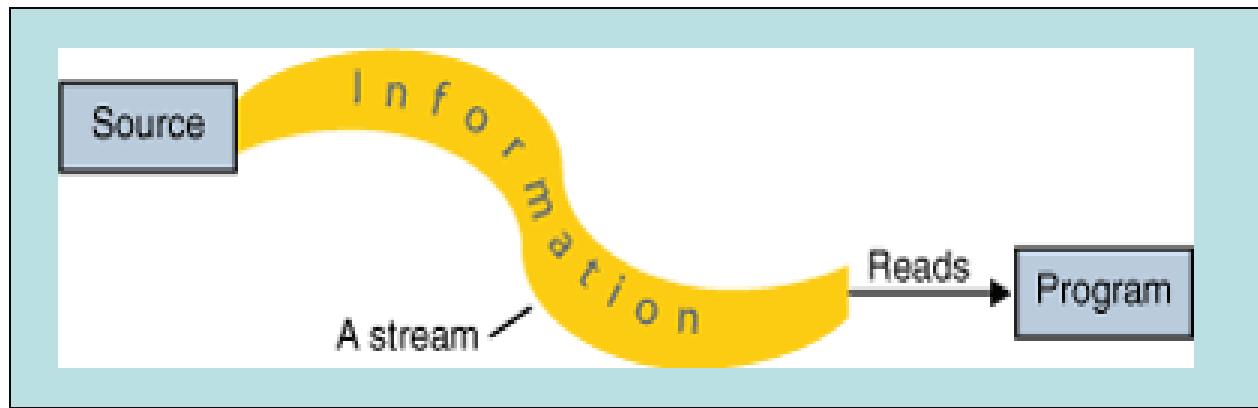
Session 8:

Learning Objectives

- **Explain Java IO**
- **Describe Streams**
 - **Byte / Character**
 - **Text /Character**

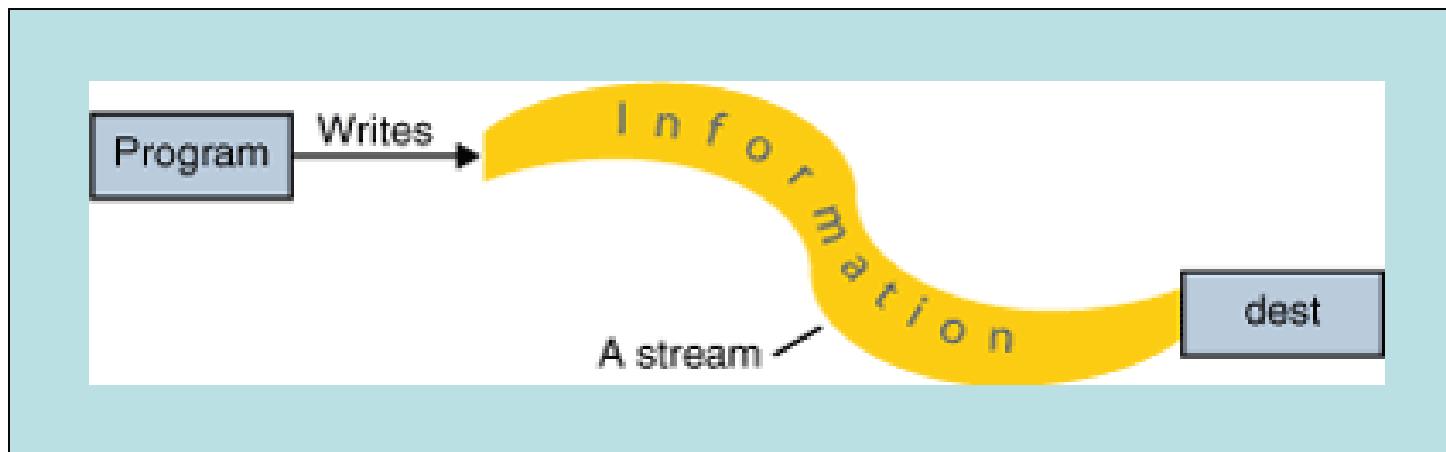
Overview of I/O Streams

To bring in information, a program opens a *stream* on an information source (a file, memory, a socket) and reads the information sequentially, as shown in the following figure.



Overview of I/O STREAMS

Similarly, a program can send information to an external destination by opening a **stream** to a destination and writing the information out sequentially, as shown in the following figure.

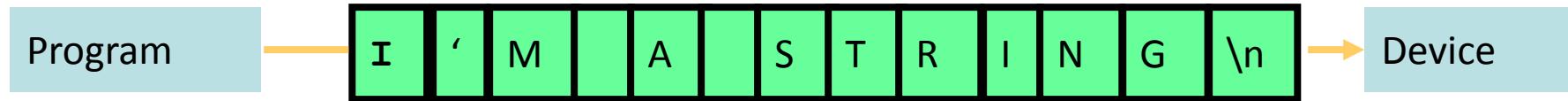


Overview of I/O streams

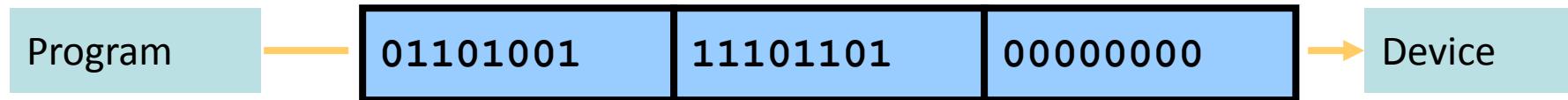
- The `java.io` package contains a collection of **stream classes** that support algorithms for reading and writing.
- To use these classes, a program needs to import the `java.io` package.
- The stream classes are divided into two class hierarchies, based on the data type (either **characters** or **bytes**) on which they operate i.e **Character Stream** and **Byte Stream**
- Java has predefined **byte** streams:
 - `System.in`
 - `System.out`
 - `System.err`

I/O Streams

- JAVA distinguishes between 2 types of streams:
- **Text – streams**, containing ‘characters’



- **Binary Streams**, containing 8 – bit information



Streams

- Streams in JAVA are Objects, of course!

Having

- 2 types of streams (text / binary) and
- 2 directions (input / output)
- Results in 4 base-classes dealing with I/O:
 1. Reader: text-input
 2. Writer: text-output
 3. InputStream: byte-input
 4. OutputStream: byte-output

InputStream

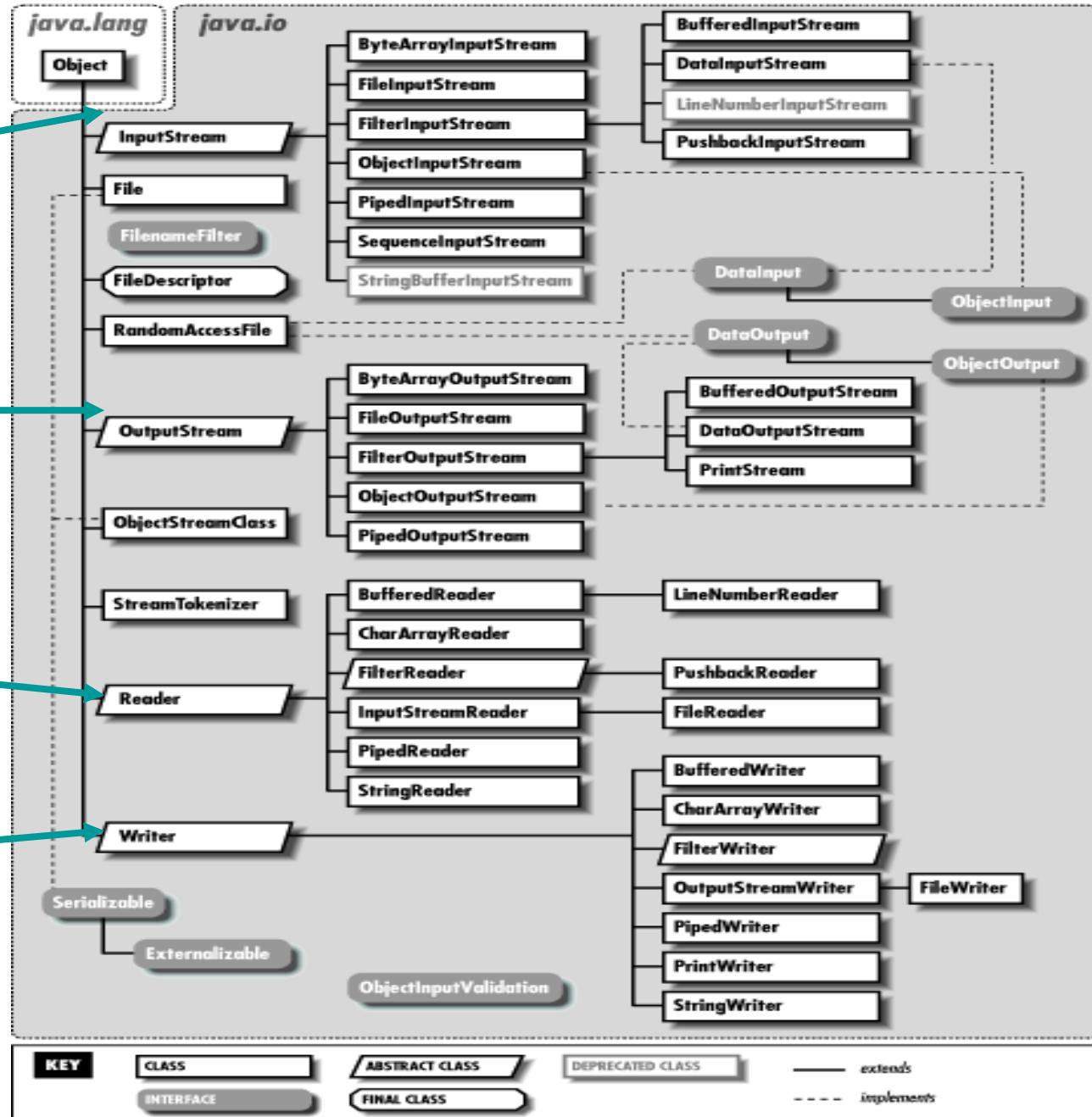
OutputStream

binary

Reader

Writer

text



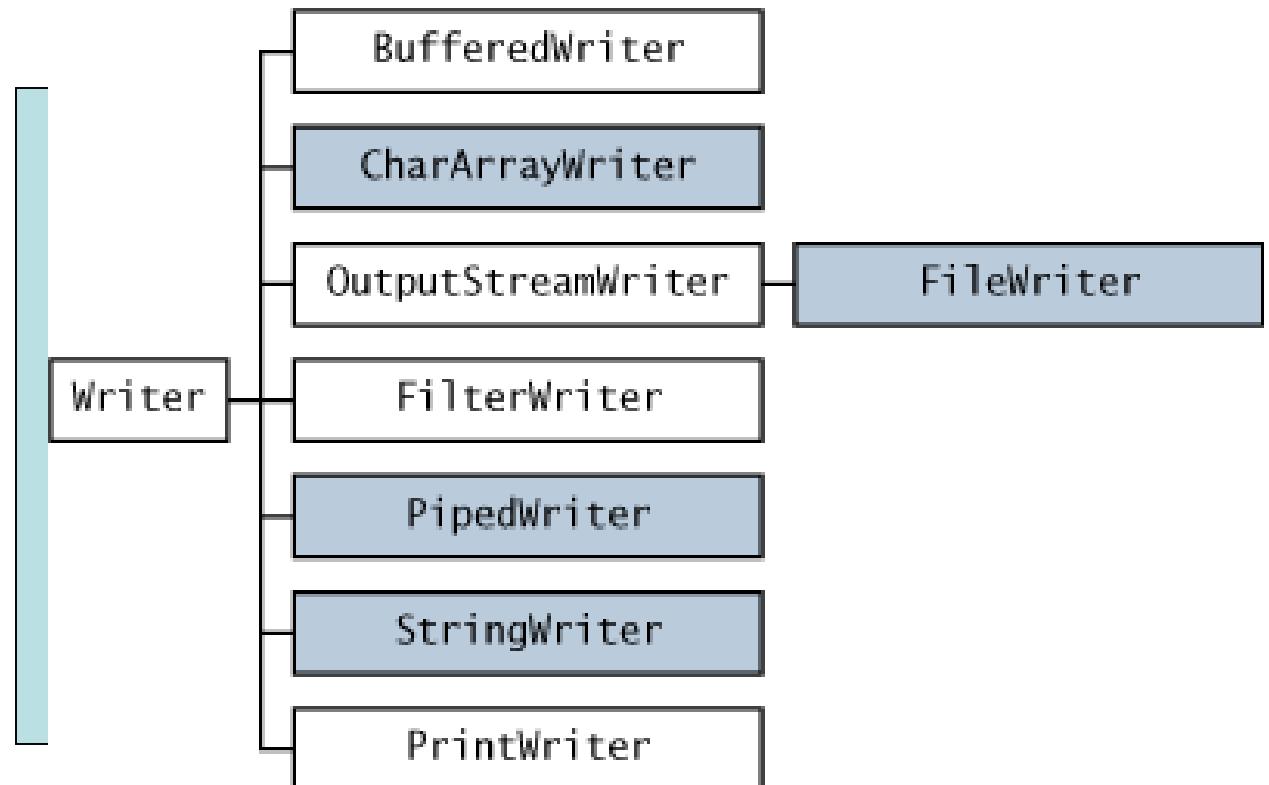
Character Streams

- Reader and Writer are the **abstract super classes** for character streams in java.io
- **Reader** provides the API and partial implementation for readers (streams that read 16-bit characters)
- **Writer** provides the API and partial implementation for writers (streams that write 16-bit characters).

Character Streams

- The following figure shows the class hierarchies for the Reader and Writer classes.

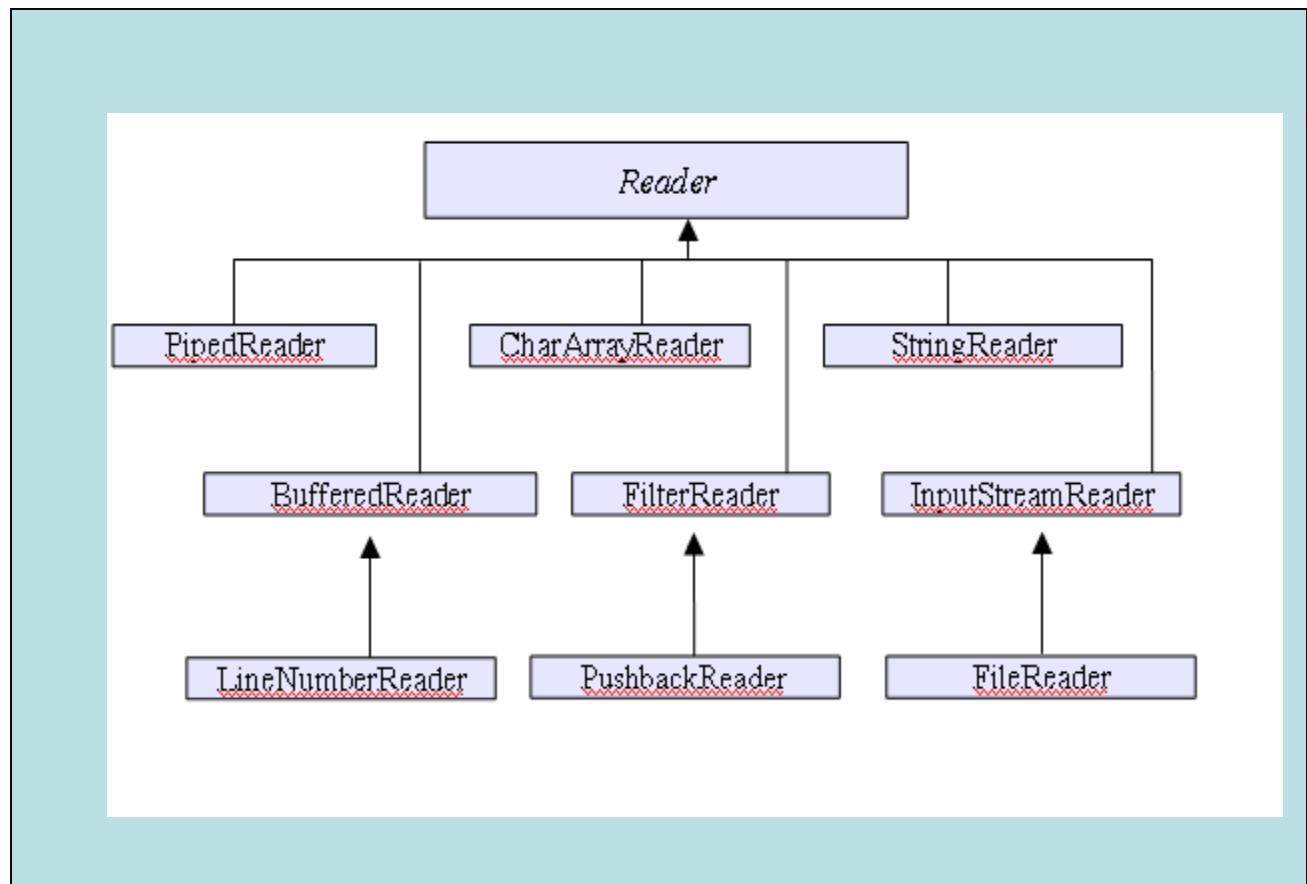
Writer Class:



Character Streams

- The following figure shows the class hierarchies for the **Reader** and **Writer** classes.

Reader class:



FileWriter class:

[<<prev](#) [next>>](#)

FileWriter class is used to write character-oriented data to the file. Sun Microsystem has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

Example of FileWriter class:

In this example, we are writing the data in the file abc.txt.

```
import java.io.*;
class Simple{
    public static void main(String args[]){
        try{
            FileWriter fw=new FileWriter("abc.txt");
            fw.write("my name is sachin");
            fw.flush();

            fw.close();
        }catch(Exception e){System.out.println(e);}
        System.out.println("success");
    }
}
```

FileReader class:

FileReader class is used to read data from the file.

Example of FileReader class:

In this example, we are reading the data from the file abc.txt file.

```
import java.io.*;
class Simple{
    public static void main(String args[])throws Exception{
        FileReader fr=new FileReader("abc.txt");
        int i;
        while((i=fr.read())!=-1)
            System.out.println((char)i);
        fr.close();
    }
}
```

Output:my name is sachin

Writing Textfiles

- Class: `FileWriter`
- Frequently used methods:

Method Summary	
abstract void	<code>close()</code> Close the stream, flushing it first.
abstract void	<code>flush()</code> Flush the stream.
void	<code>write(char[] cbuf)</code> Write an array of characters.
abstract void	<code>write(char[] cbuf, int off, int len)</code> Write a portion of an array of characters.
void	<code>write(int c)</code> Write a single character.
void	<code>write(String str)</code> Write a string.
void	<code>write(String str, int off, int len)</code> Write a portion of a string.

Writing Textfiles

- Using `FileWriter`
- It is not very convenient
- is not efficient (every character is written in a single step, invoking a huge overhead)
- Better: wrap `FileWriter` with processing streams
- `BufferedWriter`
- `PrintWriter`

Example

- Writing a textfile:

```
import java.io.*;

public class IOTest
{
    public static void main(String[] args)
    {
        try
        {
            FileWriter out = new FileWriter("test.txt");
            BufferedWriter b = new BufferedWriter(out);
            PrintWriter p = new PrintWriter(b);

            p.println("I'm a sentence in a text-file");

            p.close();
        } catch(Exception e){}
    }
}
```

- Create a stream object and associate it with a disk-file
- Give the stream object the desired functionality
- write data to the stream
- close the stream.

Wrapping Textfiles

- BufferedWriter:
- Buffers output of FileWriter, i.e. multiple characters are processed together, enhancing efficiency
- PrintWriter
- provides methods for convenient handling, e.g. println()
- (remark: the System.out.println() – method is a method of the PrintWriter-instance System.out !)

Wrapping a Writer

- A typical code segment for opening a convenient, efficient `textfile`:
 - `FileWriter out = new FileWriter("test.txt");`
 - `BufferedWriter b = new BufferedWriter(out);`
 - `PrintWriter p = new PrintWriter(b);`
- Or
- with anonymous ('unnamed') objects:
 - `PrintWriter p = new PrintWriter(new BufferedWriter(new FileWriter("test.txt")));`

Reading Textfiles

- Class: FileReader
- Frequently used Methods:

Method Summary	
abstract void	close() Close the stream.
void	mark(int readAheadLimit) Mark the present position in the stream.
boolean	markSupported() Tell whether this stream supports the mark() operation.
int	read() Read a single character.
int	read(char[] cbuf) Read characters into an array.
abstract int	read(char[] cbuf, int off, int len) Read characters into a portion of an array.
boolean	ready() Tell whether this stream is ready to be read.
void	reset() Reset the stream.
long	skip(long n) Skip characters.

(The other methods are used for positioning)

Wrapping a Reader

- Using FileReader is not very efficient.
- Better wrap it with BufferedReader:
 - `BufferedReader br =new BufferedReader(
 new FileReader("name"));`
 - Remark: BufferedReader contains the method `readLine()`, which is convenient for reading textfiles

EOF Detection

- Detecting the end of a file (EOF):
- Usually amount of data to be read is not known
- Reading methods return 'impossible' value if end of file is reached
- Example:
 - FileReader.read returns `-1`
 - BufferedReader.readLine() returns '`null`'
- Typical code for EOF detection:
- ```
while ((c = myReader.read() != -1){ // read and
check c
 ...do something with c
}
```

# Example

```
import java.io.*;
public class IOTest1
{
 public static void main(String[] args)
 {
 try{
 BufferedReader myInput = new BufferedReader(new
 FileReader("IOTest1.java"));

 BufferedWriter myOutput = new BufferedWriter(new
 FileWriter("Test1.txt"));
 int c;
 while((c=myInput.read()) != -1)
 myOutput.write(c);
 myInput.close();
 myOutput.close();
 }catch(IOException e){}
 }
}
```

```
import java.io.*;
public class BReader {

 public static void main(String[] args) throws
 IOException {
 BufferedReader br=new BufferedReader(new
 FileReader("abcd.txt"));

 String line=br.readLine();
 while(line!=null){
 System.out.println(line);
 line=br.readLine();
 }
 br.close();
 }
}
```

# Byte Streams

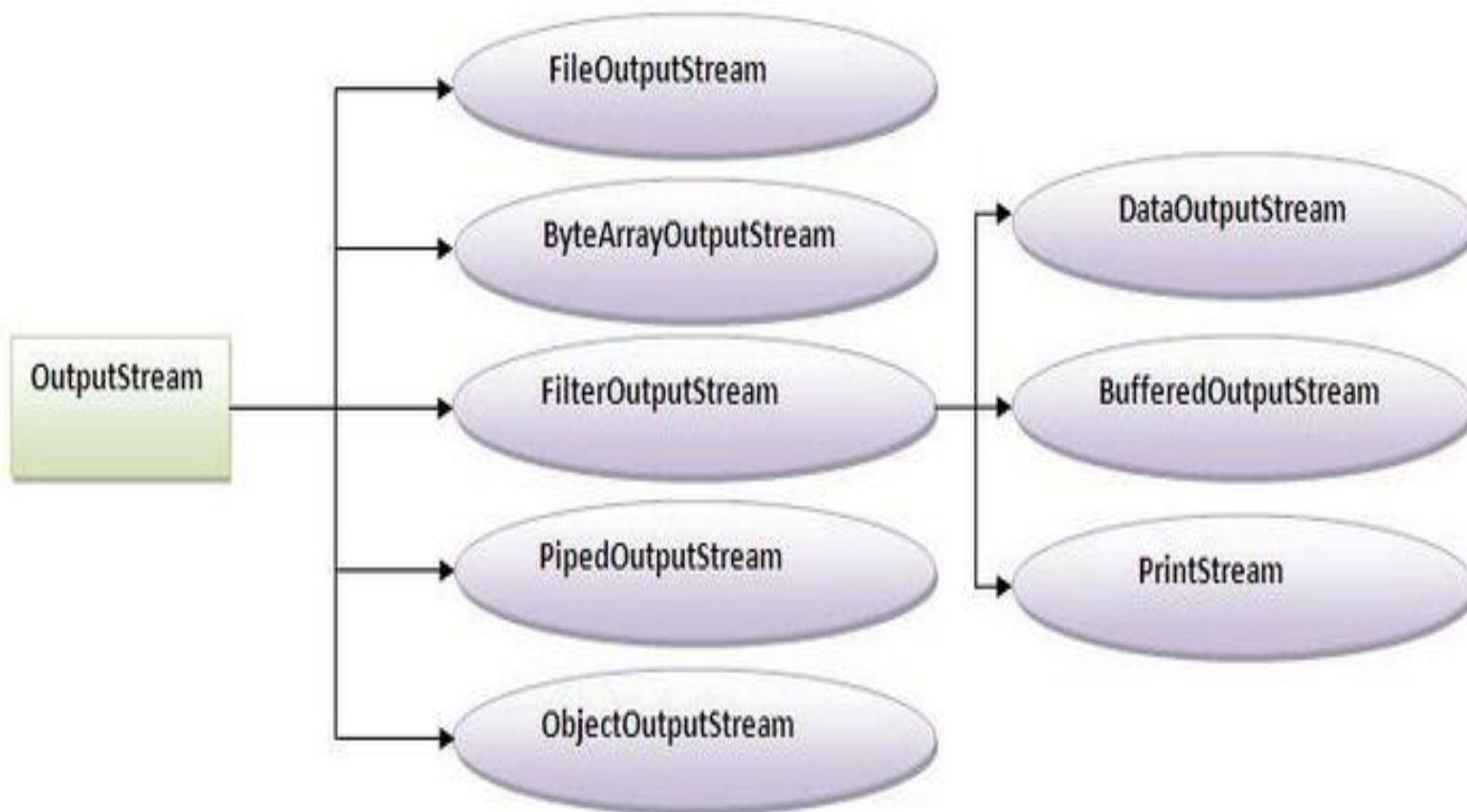
- To read and write 8-bit bytes, programs should use the byte streams, descendants of [InputStream](#) and [OutputStream](#).
- [InputStream](#) and [OutputStream](#) provide the API and partial implementation for input streams (streams that read 8-bit bytes) and output streams (streams that write 8-bit bytes).
- These streams are typically used to read and write binary data such as images and sounds.
- Two of the byte stream classes, **ObjectInputStream** and **ObjectOutputStream**, are used for object serialization.

## OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

### Commonly used methods of OutputStream class

| Method                                                  | Description                                                     |
|---------------------------------------------------------|-----------------------------------------------------------------|
| <b>1) public void write(int) throws IOException:</b>    | is used to write a byte to the current output stream.           |
| <b>2) public void write(byte[]) throws IOException:</b> | is used to write an array of byte to the current output stream. |
| <b>3) public void flush() throws IOException:</b>       | flushes the current output stream.                              |
| <b>4) public void close() throws IOException:</b>       | is used to close the current output stream.                     |

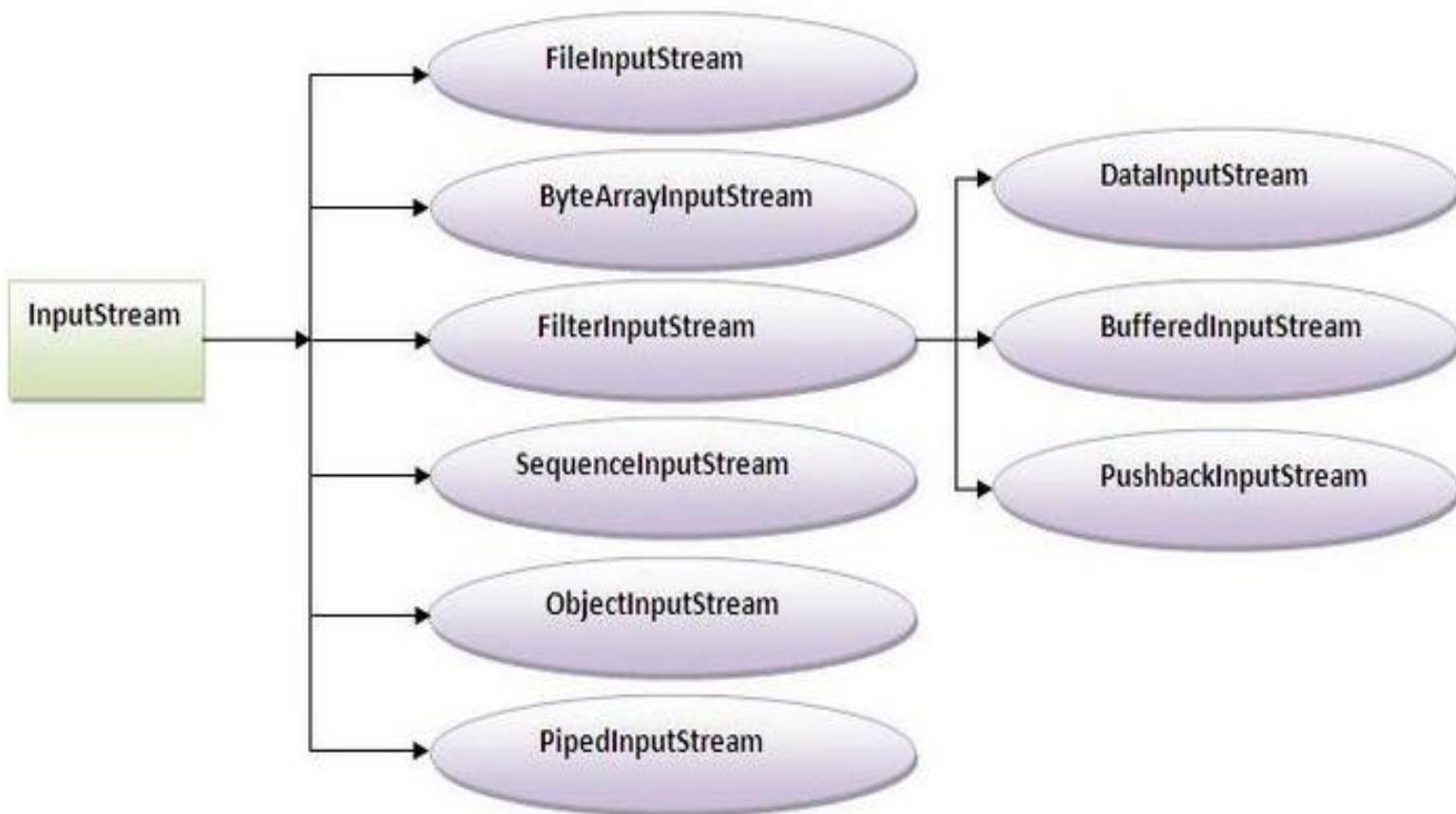


## InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

### Commonly used methods of InputStream class

| Method                                                  | Description                                                                                |
|---------------------------------------------------------|--------------------------------------------------------------------------------------------|
| <b>1) public abstract int read()throws IOException:</b> | reads the next byte of data from the input stream. It returns -1 at the end of file.       |
| <b>2) public int available()throws IOException:</b>     | returns an estimate of the number of bytes that can be read from the current input stream. |
| <b>3) public void close()throws IOException:</b>        | is used to close the current input stream.                                                 |



## FileOutputStream class:

A FileOutputStream is an output stream for writing data to a file.

If you have to write primitive values then use FileOutputStream.Instead, for character-oriented data, prefer FileWriter.But you can write byte-oriented as well as character-oriented data.

## Example of FileOutputStream class:

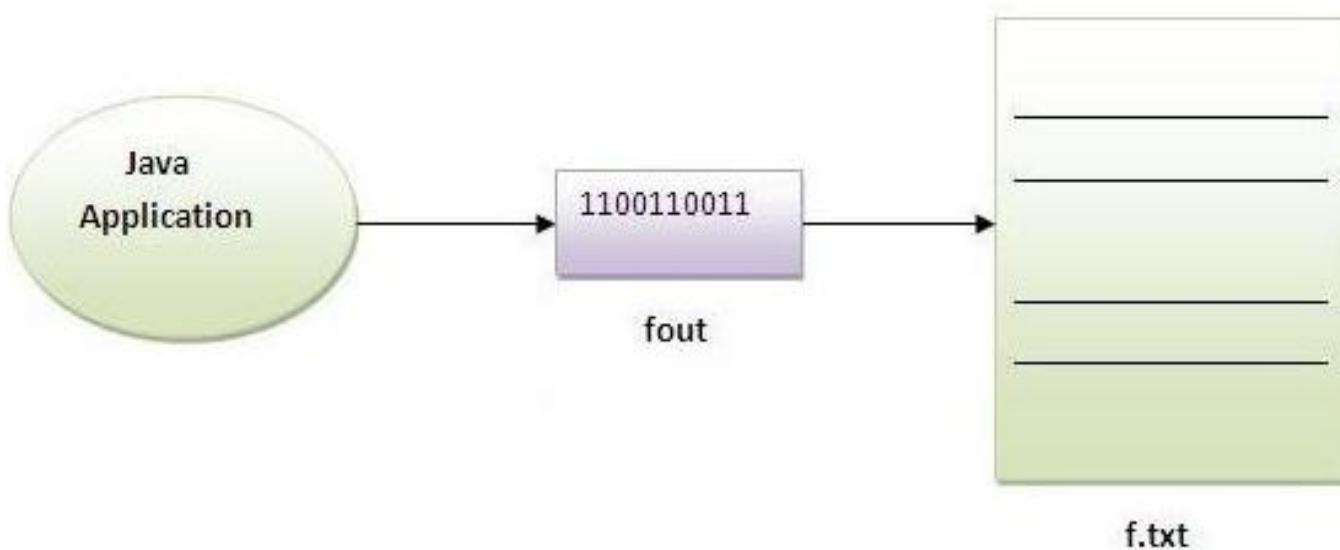
```
//<i>Simple program of writing data into the file</i>

import java.io.*;
class Test{
 public static void main(String args[]){
 try{
 FileOutputStream fout=new FileOutputStream("abc.txt");
 String s="Sachin Tendulkar is my favourite player";

 byte b[]={s.getBytes()};
 fout.write(b);

 fout.close();

 System.out.println("success...");
 }catch(Exception e){System.out.println(e);}
 }
}
```



## FileInputStream class:

A FileInputStream obtains input bytes from a file. It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

It should be used to read byte-oriented data. For example, to read image etc.

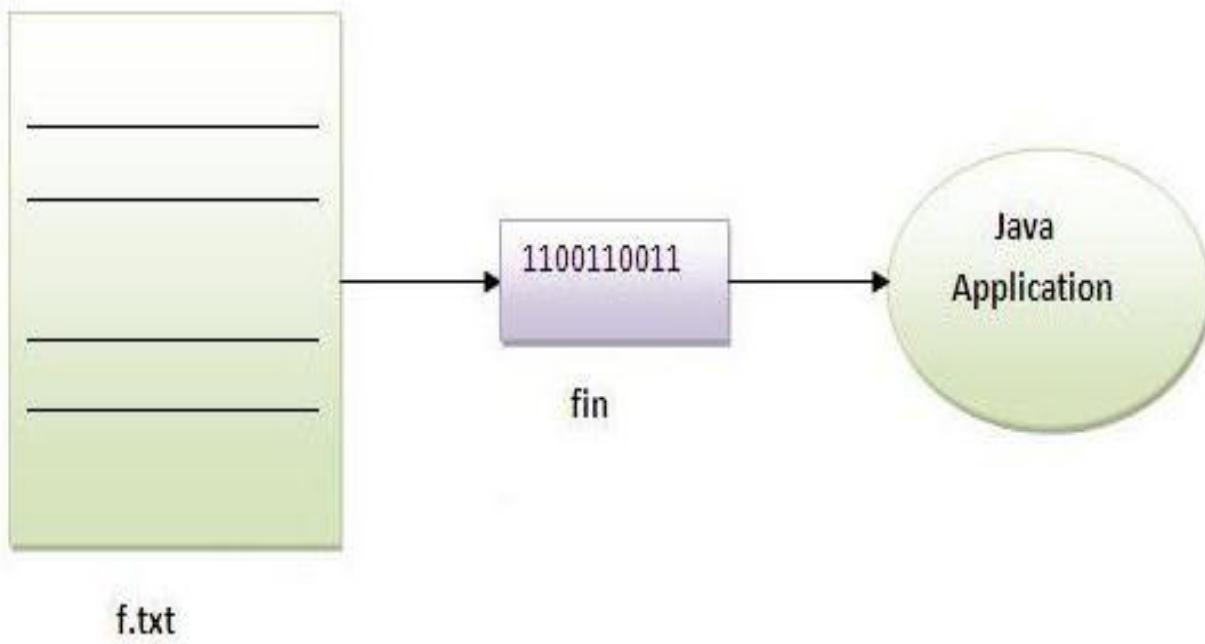
## Example of FileInputStream class:

```
//<i>Simple program of reading data from the file</i>

import java.io.*;
class SimpleRead{
public static void main(String args[]){
try{
FileInputStream fin=new FileInputStream("abc.txt");
int i;
while((i=fr.read())!=-1)
System.out.println((char)i);

fin.close();
}catch(Exception e){system.out.println(e);}
}
}
```

<strong>Output:</strong> Sachin is my favourite player.



## Example of Reading the data of current java file and writing it into another file

We can read the data of any file using the `FileInputStream` class whether it is java file, image file, video file etc. In this example, we are reading the data of `C.java` file and writing it into another file `M.java`.

```
import java.io.*;

class C{
public static void main(String args[])throws Exception{

FileInputStream fin=new FileInputStream("C.java");
FileOutputStream fout=new FileOutputStream("M.java");

int i=0;
while((i=fin.read())!=-1){
fout.write((byte)i);
}

fin.close();
}
}
```

## BufferedOutputStream class:

[<<prev](#) [next>](#)

BufferedOutputStream used an internal buffer. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

### Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```
import java.io.*;
class Test{
 public static void main(String args[])throws Exception{
 FileOutputStream fout=new FileOutputStream("f1.txt");
 BufferedOutputStream bout=new BufferedOutputStream(fout);

 String s="Sachin is my favourite player";
 byte b[]={s.getBytes()};
 bout.write(b);

 bout.flush();
 bout.close();
 System.out.println("success");
 }
}
```

## Example of BufferedInputStream class:

```
//<i>Simple program of reading data from the file using buffer</i>

import java.io.*;
class SimpleRead{
 public static void main(String args[]){
 try{

 FileInputStream fin=new FileInputStream("f1.txt");
 BufferedInputStream bin=new BufferedInputStream(fin);
 int i;
 while((i=bin.read())!=-1)
 System.out.println((char)i);

 fin.close();
 }catch(Exception e){System.out.println(e); }
 }
}
```

<strong>Output:</strong>Sachin is my favourite player

# Reading data from keyboard:

There are many ways to read data from the keyboard. For example:

- InputStreamReader
- Console
- Scanner
- DataInputStream etc.

## **InputStreamReader class:**

InputStreamReader class can be used to read data from keyboard. It performs two tasks:

- connects to input stream of keyboard
- converts the byte-oriented stream into character-oriented stream

## **BufferedReader class:**

BufferedReader class can be used to read data line by line by readLine() method.

## **Example of reading data from keyboard by InputStreamReader and BufferdReader class:**

In this example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

```
//Program of reading data

import java.io.*;
class G5{
public static void main(String args[])throws Exception{

InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);

System.out.println("Enter ur name");
String name=br.readLine();
System.out.println("Welcome "+name);
}
}
```

Output:Enter ur name  
Amit  
Welcome Amit

In this example, we are reading and printing the data until the user prints stop.

```
import java.io.*;
class G5{
public static void main(String args[]) throws Exception{

InputStreamReader r=new InputStreamReader(System.in);
BufferedReader br=new BufferedReader(r);

String name="";
while(name.equals("stop")){
System.out.println("Enter data: ");
name=br.readLine();
System.out.println("data is: "+name);
}

br.close();
r.close();
}
}
```

Output: Enter data: Amit  
data is: Amit  
Enter data: 10  
data is: 10  
Enter data: stop

# Console class (I/O)

The Console class can be used to get input from the keyboard.

## How to get the object of Console class?

System class provides a static method named `console()` that returns the unique instance of Console class.

### Syntax:

```
public static Console console(){}
```

## Commonly used methods of Console class:

- 1) public String readLine():** is used to read a single line of text from the console.
- 2) public String readLine(String fmt, Object... args):** it provides a formatted prompt then reads the single line of text from the console.
- 3) public char[] readPassword():** is used to read password that is not being displayed on the console.
- 4) public char[] readPassword(String fmt, Object... args):** it provides a formatted prompt then reads the password that is not being displayed on the console.

## Example of Console class that reads name of user:

```
import java.io.*;
class A{
public static void main(String args[]){
 Console c=System.console();
 System.out.println("Enter ur name");
 String n=c.readLine();
 System.out.println("Welcome "+n);
}
}
```

## java.util.Scanner class:

[!\[\]\(9a3e26efbcaf223bb76dca1e82607996\_img.jpg\) <<prev](#) [!\[\]\(156e4e3560e4a40671124e7edfbf8561\_img.jpg\) next>>](#)

There are various ways to read input from the keyboard, the java.util.Scanner class is one of them. The Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

### Commonly used methods of Scanner class:

There is a list of commonly used Scanner class methods:

- **public String next():** it returns the next token from the scanner.
- **public String nextLine():** it moves the scanner position to the next line and returns the value as a string.
- **public byte nextByte():** it scans the next token as a byte.
- **public short nextShort():** it scans the next token as a short value.
- **public int nextInt():** it scans the next token as an int value.
- **public long nextLong():** it scans the next token as a long value.
- **public float nextFloat():** it scans the next token as a float value.
- **public double nextDouble():** it scans the next token as a double value.

## Example of java.util.Scanner class:

Let's see the simple example of the Scanner class which reads the int, string and double value as an input:

```
import java.util.Scanner;
class ScannerTest{
 public static void main(String args[]){

 Scanner sc=new Scanner(System.in);

 System.out.println("Enter your rollno");
 int rollno=sc.nextInt();
 System.out.println("Enter your name");
 String name=sc.next();
 System.out.println("Enter your fee");
 double fee=sc.nextDouble();

 System.out.println("Rollno:"+rollno+" name:"+name+" fee:"+fee);

 }
}
```

## The scanner class

```
import java.util.*;
import java.io.File;
import java.io.FileNotFoundException;
public class scanIn {
 public static void main(String[] args) {
 String first, last;
 int ssn;
 try{
 Scanner sc = new Scanner(new File("input.txt"));
 while (sc.hasNext()) {
 first = sc.next();
 last = sc.next();
 ssn = sc.nextInt();
 System.out.println("First: " + first + "\nLast: " + last + "\nSSN: " + ssn);
 }
 }catch (FileNotFoundException e){
 System.out.println(e);
 } //end catch
 } //end main
} // end class
```

## DataInputStream

```
/*Example using DataInputStream.*/
import java.io.DataInputStream;
public class InputOutput
{
 public static void main(String[] args)
 {
 try
 {
 DataInputStream dis = new DataInputStream(System.in);
 System.out.println("Enter First Number");
 int a = Integer.parseInt(dis.readLine());
 System.out.println("Enter Second Number");
 int b = Integer.parseInt(dis.readLine());
 int sum = a+b;
 System.out.println("Sum is "+sum);
 }
 catch (Exception e)
 {
 e.printStackTrace();
 }
 }
}
```

Output:

```
Enter First Number
2
Enter Second Number
4
Sum is 6
```

**Next....**

**Multithreading**

# **Session 9:**

## **Learning Objectives**

- Explain Multithreading

# Multithreading

## Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

### 1)Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

### 2)Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.
- **Note:**At least one process is required for each thread.

# Threads

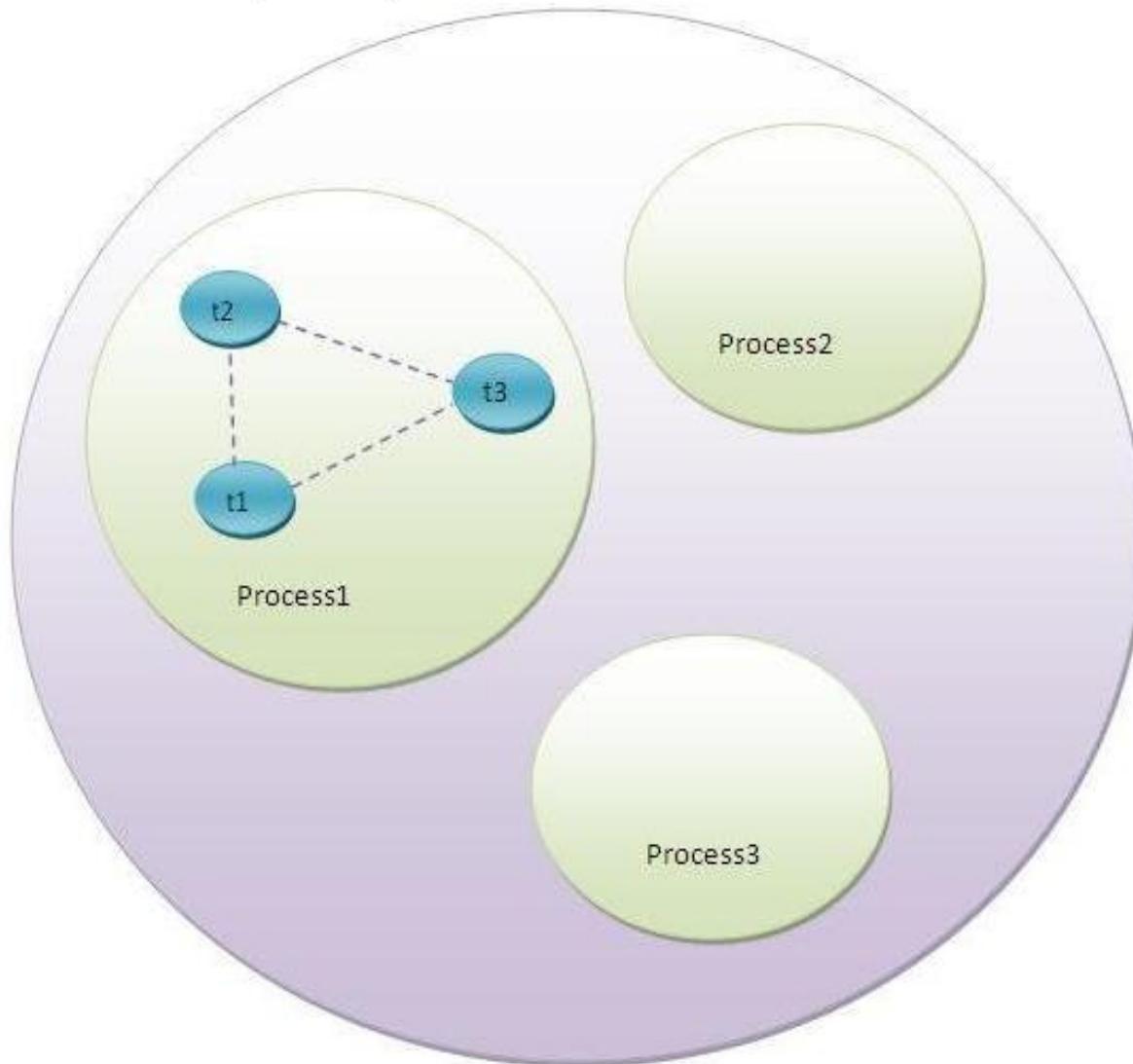
- Threads are lightweight processes as the overhead of switching between threads is less
- They can be easily spawned
- The Java Virtual Machine spawns a thread when your program is run called the Main Thread

## Why do we need threads?

- To enhance parallel processing
- To increase response to the user
- To utilize the idle time of the CPU
- Prioritize your work depending on priority

## What is Thread?

A thread is a lightweight subprocess, a smallest unit of processing. It is a separate path of execution. It shares the memory area of process.



# Life cycle of a Thread (Thread States)

A thread can be in one of the five states in the thread. According to sun, there is only 4 states new, runnable, non-runnable and terminated. There is no running state. But for better understanding the threads, we are explaining it in the 5 states. The life cycle of the thread is controlled by JVM. The thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

## ▶ Life cycle of a thread

- ▶ New
- ▶ Runnable
- ▶ Running
- ▶ Non-Runnable (Blocked)
- ▶ Terminated

## 1)New

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

## 2)Runnable

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

## 3)Running

The thread is in running state if the thread scheduler has selected it.

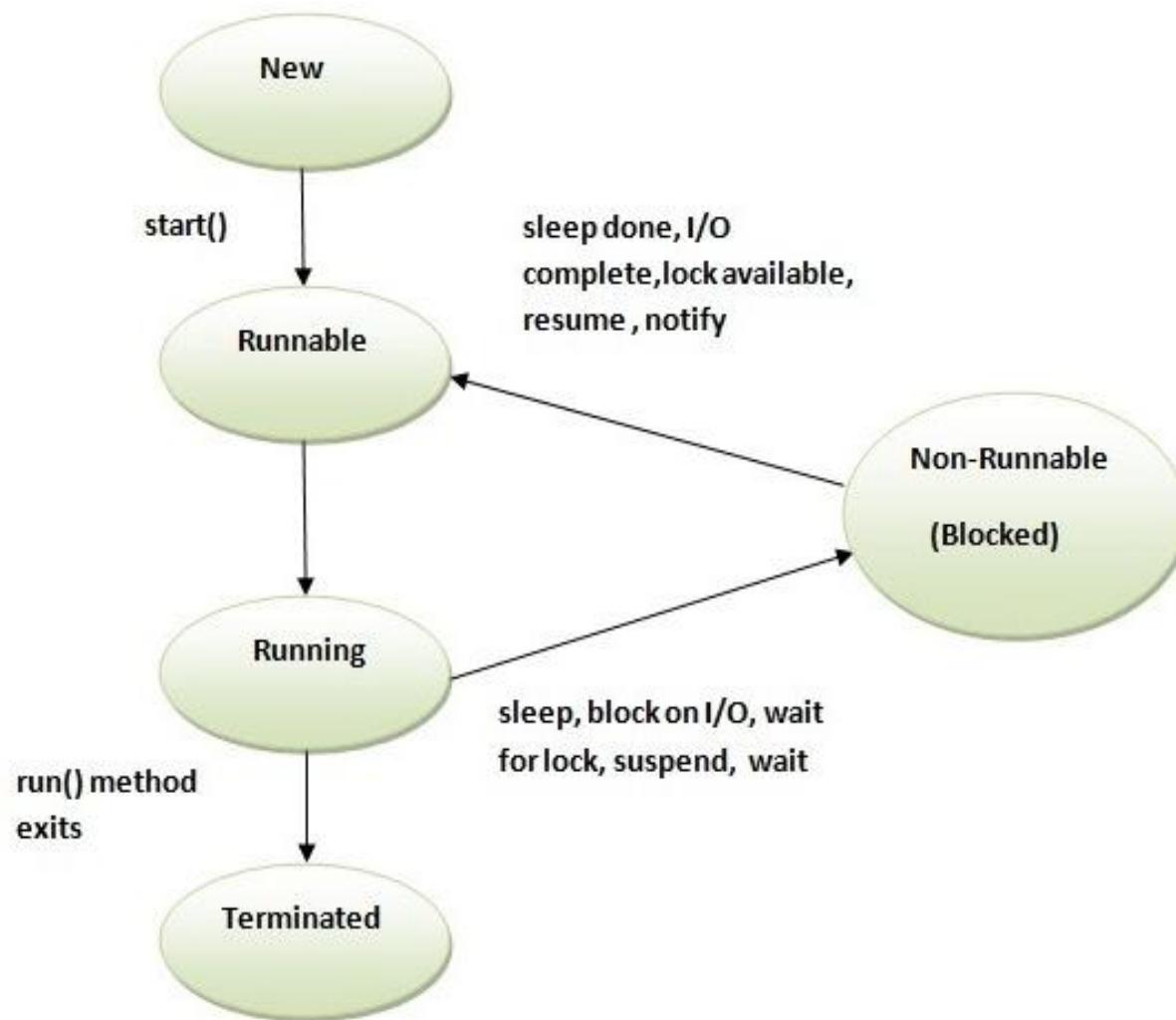
## 4)Non-Runnable (Blocked)

This is the state when the thread is still alive, but is currently not eligible to run.

## 5)Terminated

A thread is in terminated or dead state when its run() method exits.

## Thread States



# How to create thread:

There are two ways to create a thread:

1. By extending Thread class
  2. By implementing Runnable interface.
- 

## Thread class:

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

## Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

---

## Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long milliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long milliseconds):** waits for a thread to die for the specified milliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(deprecated).
16. **public void resume():** is used to resume the suspended thread(deprecated).
17. **public void stop():** is used to stop the thread(deprecated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

## Runnable interface:

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().

1. **public void run()**: is used to perform action for a thread.
- 

## Starting a thread:

**start()** method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

## 1)By extending Thread class:

```
class Multi extends Thread{
 public void run(){
 System.out.println("thread is running...");
 }
 public static void main(String args[]){
 Multi t1=new Multi();
 t1.start();
 }
}
```

**Output:** thread is running...

## Who makes your class object as thread object?

**Thread class constructor** allocates a new thread object. When you create object of Multi class, your class constructor is invoked(provided by Compiler) from where Thread class constructor is invoked(by super() as first statement). So your Multi class object is thread object now.

## Example

```
class MyThread extends Thread{

 public void run(){ // job of thread
 for (int i=0;i<10;i++)
 System.out.println("Child Thread!!");
 }
 public static void main(String[] args){
 MyThread t=new MyThread(); // Thread instantiation
 t.start(); // starting a thread

 for(int i=0;i<10;i++)
 System.out.println("Main Thread!!");
 }
}
```

## 2)By implementing the Runnable interface:

```
class Multi3 implements Runnable{
 public void run(){
 System.out.println("thread is running...");
 }

 public static void main(String args[]){
 Multi3 m1=new Multi3();
 Thread t1 =new Thread(m1);
 t1.start();
 }
}
```

**Output:**thread is running...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitly create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

# Example

```
class MyRunnable implements Runnable{
 public void run(){ // job of thread
 for (int i=0;i<10;i++)
 System.out.println("Child Thread!!");
 }
 public static void main(String[] args){
 MyRunnable r= new MyRunnable(); // MyRunnable instantiation
 Thread t=new Thread(r); // thread instantiation
 t.start(); // starting a thread

 for(int i=0;i<10;i++)
 System.out.println("Main Thread!!");
 }
}
```

## The Thread Scheduler:

- The thread scheduler is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- Only one thread at a time can run in a single process.
- The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

### What is the difference between preemptive scheduling and time slicing?

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

# Sleeping a thread - sleep() method

- used to sleep a thread for specific time

## Syntax of sleep() method:

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long milliseconds) throws InterruptedException
- public static void sleep(long milliseconds, int nanos) throws InterruptedException

```
//<i>Program of sleep() method</i>

class Multi extends Thread{
 public void run(){
 for(int i=1;i<5;i++){
 try{Thread.sleep(500); }catch(InterruptedException e){System.out.println(e);}
 System.out.println(i);
 }
 }
 public static void main(String args[]){
 Multi t1=new Multi();
 Multi t2=new Multi();

 t1.start();
 t2.start();
 }
}
```

# Can we start a thread twice?

[\*\*<<prev\*\*](#) [\*\*next>>\*\*](#)

No. After starting a thread, it can never be started again. If you do so, an `IllegalThreadStateException` is thrown. For Example:

```
class Multi extends Thread{
 public void run(){
 System.out.println("running...");
 }
 public static void main(String args[]){
 Multi t1=new Multi();
 t1.start();
 t1.start();
 }
}
```

Output: running  
Exception in thread "main" java.lang.IllegalThreadStateException

# What if we call run() method directly instead start() method?

[<<prev](#) [next>](#)

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
class Multi extends Thread{
 public void run(){
 System.out.println("running...");
 }
 public static void main(String args[]){
 Multi t1=new Multi();
 t1.run(); //fine, but does not start a separate call stack
 }
}
```

<strong>Output:</strong>running...

## Problem ---run() directly

```
class Multi extends Thread{
 public void run(){
 for(int i=1;i<5;i++){
 try{Thread.sleep(500); }catch(InterruptedException e){System.out.println(e); }
 System.out.println(i);
 }
 }

 public static void main(String args[]){
 Multi t1=new Multi();
 Multi t2=new Multi();

 t1.run();
 t2.run();
 }
}
```

---

**Output :**

```
2
3
4
5
1
2
3
4
5
```

## The join() method:

[prev](#) [next](#)

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

### Syntax:

```
public void join()throws InterruptedException
```

```
public void join(long miliseconds) throws InterruptedException
```

//<b><i>Example of join() method</i></b>

```
class Multi extends Thread{
 public void run(){
 for(int i=1;i<=5;i++){
 try{
 Thread.sleep(500);
 }catch(Exception e){System.out.println(e);}
 System.out.println(i);
 }
 }

 public static void main(String args[]){
 Multi t1=new Multi();
 Multi t2=new Multi();
 Multi t3=new Multi();
 t1.start();
 try{
 t1.join();
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

As you can see in the above example, when t1 completes its task then t2 and t3 starts executing.

```
//<i>Example of join(long milliseconds) method</i>
```

```
class Multi extends Thread{
 public void run(){
 for(int i=1;i<=5;i++){
 try{
 Thread.sleep(500);
 }catch(Exception e){System.out.println(e);}
 System.out.println(i);
 }
 }

 public static void main(String args[]){
 Multi t1=new Multi();
 Multi t2=new Multi();
 Multi t3=new Multi();
 t1.start();
 try{
 t1.join(1500);
 }catch(Exception e){System.out.println(e);}

 t2.start();
 t3.start();
 }
}
```

## Naming a thread:



The Thread class provides methods to change and get the name of a thread.

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

## Example of naming a thread:

```
class Multi6 extends Thread{
 public void run(){
 System.out.println("running...");
 }
 public static void main(String args[]){
 Multi6 t1=new Multi6();
 Multi6 t2=new Multi6();
 System.out.println("Name of t1:"+t1.getName());
 System.out.println("Name of t2:"+t2.getName());

 t1.start();
 t2.start();

 t1.setName("Sonoo Jaiswal");
 System.out.println("After changing name of t1:"+t1.getName());
 }
}
```

## The current Thread() method:

The currentThread() method returns a reference to the currently executing thread object.

### Syntax of currentThread() method:

- **public static Thread currentThread():** returns the reference of currently running thread.

### Example of currentThread() method:

```
class Multi6 extends Thread{
 public void run(){
 System.out.println(Thread.currentThread().getName());
 }
}

public static void main(String args[]){
 Multi6 t1=new Multi6();
 Multi6 t2=new Multi6();

 t1.start();
 t2.start();
}
```

**Output:**  
Thread-0  
Thread-1

## Priority of a Thread (Thread Priority):

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

### 3 constants defined in Thread class:

1. public static int MIN\_PRIORITY
2. public static int NORM\_PRIORITY
3. public static int MAX\_PRIORITY

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

## Example of priority of a Thread:

```
class Multi10 extends Thread{
 public void run(){
 System.out.println("running thread name is:"+Thread.currentThread().getName());
 System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
 }
 public static void main(String args[]){
 Multi10 m1=new Multi10();
 Multi10 m2=new Multi10();
 m1.setPriority(Thread.MIN_PRIORITY);
 m2.setPriority(Thread.MAX_PRIORITY);
 m1.start();
 m2.start();
 }
}
```

**Output:**running thread name is:Thread-0  
running thread priority is:10  
running thread name is:Thread-1  
running thread priority is:1

# Daemon Thread

There are two types of threads user thread and daemon thread. The daemon thread is a service provider thread. It provides services to the user thread. Its life depends on the user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

## Points to remember for Daemon Thread:

- It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
- Its life depends on user threads.
- It is a low priority thread.

## Why JVM terminates the daemon thread if there is no user thread remaining?

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

## Methods for Daemon thread:

The `java.lang.Thread` class provides two methods related to daemon thread

- **`public void setDaemon(boolean status)`:** is used to mark the current thread as daemon thread or user thread.
- **`public boolean isDaemon()`:** is used to check that current is daemon.

## Simple example of Daemon thread

```
class MyThread extends Thread{
 public void run(){
 System.out.println("Name: "+Thread.currentThread().getName());
 System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }

 public static void main(String[] args){
 MyThread t1=new MyThread();
 MyThread t2=new MyThread();
 t1.setDaemon(true);

 t1.start();
 t2.start();
 }
}
```

**Output:**  
Name: thread-0  
Daemon: true  
Name: thread-1  
Daemon: false



**Note:** If you want to make a user thread as Daemon, it must not be started otherwise it will throw `IllegalThreadStateException`.



**Note:** If you want to make a user thread as Daemon, it must not be started otherwise it will throw `IllegalThreadStateException`.

```
class MyThread extends Thread{
 public void run(){
 System.out.println("Name: "+Thread.currentThread().getName());
 System.out.println("Daemon: "+Thread.currentThread().isDaemon());
 }

 public static void main(String[] args){
 MyThread t1=new MyThread();
 MyThread t2=new MyThread();
 t1.start();
 t1.setDaemon(true); //will throw exception here
 t2.start();
 }
}
```

**Output:**exception in thread main: java.lang.IllegalThreadStateException

# Synchronization

Synchronization is the capability of control the access of multiple threads to any shared resource. Synchronization is better in case we want only one thread can access the shared resource at a time.

## Why use Synchronization?

The synchronization is mainly used to

1. To prevent thread interference.
  2. To prevent consistency problem.
- 

## Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

## Understanding the problem without Synchronization

```
class Table{
 void printTable(int n){ //method not synchronized
 for(int i=1;i<=5;i++){
 System.out.println(n*i);
 try{
 Thread.sleep(400);
 }catch(Exception e){System.out.println(e);}
 }
 }

 class MyThread1 extends Thread{
 Table t;
 MyThread1(Table t){
 this.t=t;
 }
 public void run(){
 t.printTable(5);
 }
 }

 class MyThread2 extends Thread{
 Table t;
 MyThread2(Table t){
 this.t=t;
 }
 public void run(){
 t.printTable(100);
 }
 }
}
```

```
class Use{
 public static void main(String args[]){
 Table obj = new Table(); //only one object
 MyThread1 t1=new MyThread1(obj);
 MyThread2 t2=new MyThread2(obj);
 t1.start();
 t2.start();
 }
}
```

Output:

5  
100  
10  
200  
....  
...

USE --- synchronized void printTable()

## Example 2: synchronization

```
public class Greeting {
 public synchronized void wish(String name){
 for(int i=0;i<10;i++){
 System.out.println("Good Morning:");
 try{
 Thread.sleep(5000);
 }catch(InterruptedException e){}
 System.out.println(name);
 }
 }
}
```

```
public class MyThread extends Thread{

 Greeting g;
 String name;

 public MyThread(Greeting g, String name) {
 this.g=g;
 this.name=name;
 }
 @Override
 public void run(){
 g.wish(name);
 }
}
```

```
public class SynchronizedDemo {
 public static void main(String[] args) {
 Greeting g=new Greeting();
 MyThread t1=new MyThread(g,"DAC");
 MyThread t2=new MyThread(g, "DSSD");
 t1.start();
 t2.start();
 }
}
```

## Output:

## Example 3: synchronization

```
public class Display {
 public synchronized void dispn(){
 for(int i=1;i<=10;i++){
 System.out.println(i);
 }
 try{
 Thread.sleep(3000);
 }catch(InterruptedException e){}
 }
 public synchronized void dispc(){
 for(int i=65;i<75;i++){
 System.out.println((char)i);
 }
 try{
 Thread.sleep(3000);
 }catch(InterruptedException e){}
 }
}

public class SyncDemo {
 public static void main(String[] args) {
 Display d=new Display();

 Mythread1 t1=new Mythread1(d);
 Mythread2 t2=new Mythread2(d);
 t1.start();
 t2.start();
 }
}
```

```
public class Mythread1 extends Thread{
 Display d;

 public Mythread1(Display d) {
 this.d = d;
 }
 public void run(){
 d.dispn();
 }
}

class Mythread2 extends Thread{
 Display d;

 public Mythread2(Display d) {
 this.d = d;
 }
 public void run(){
 d.dispc();
 }
}
```

Output:

```
1
2
3
4
5
6
7
8
9
10
A
B
C
D
E
F
G
H
I
J
```

# Program of synchronized block

```
class Table{

void printTable(int n){
 synchronized(this){ //synchronized block
 for(int i=1;i<=5;i++){
 System.out.println(n*i);
 try{
 Thread.sleep(400);
 }catch(Exception e){System.out.println(e);}
 }
 }
}//end of the method

}
```

# Inter-thread communication in Java

[\*\*<<prev\*\*](#) [\*\*next>>\*\*](#)

**Inter-thread communication or Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed. It is implemented by following methods of **Object class**:

- `wait()`
- `notify()`
- `notifyAll()`

## 1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

| Method                                                          | Description                             |
|-----------------------------------------------------------------|-----------------------------------------|
| public final void wait()throws InterruptedException             | waits until object is notified.         |
| public final void wait(long timeout)throws InterruptedException | waits for the specified amount of time. |

## 2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

```
public final void notify()
```

## 3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

```
public final void notifyAll()
```

## Example : Inter thread communication

```
class Customer{
int amount=10000;
synchronized void withdraw (int amount){
System.out.println("going to withdraw...");

if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}
```

```
class Test{
public static void main(String args[]){
Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
.start();
new Thread(){
public void run(){c.deposit(10000);}
.start();
}
}
```

### Output:

going to withdraw...  
Less balance; waiting for deposit...  
going to deposit...  
deposit completed...  
withdraw completed...

## Example : Deadlock

```
public class A {

 public synchronized void m1(B b){
 System.out.println("Thread 1 starts execution of
m1() ");
 try{
 Thread.sleep(5000);
 }catch(InterruptedException ie){}

 b.dead();
 }

 public synchronized void dead(){
 System.out.println("Inside A, dead()");
 }
}

public class Deadlock extends Thread {
 A a=new A();
 B b=new B();

 public void fun(){
 this.start();
 a.m1(b); //main thread
 }
 public void run(){
 b.m2(a); // child thread
 }
 public static void main(String[] args) {
 Deadlock d=new Deadlock();
 d.fun();
 }
}
```

```
public class B {
 public synchronized void m2(A a){
 System.out.println("Thread 2 starts execution of
m1() ");
 try{
 Thread.sleep(5000);
 }catch(InterruptedException ie){}

 a.dead();
 }

 public synchronized void dead(){
 System.out.println("Inside B, dead()");
 }
}
```

### Output: (deadlock)

Thread 1 starts execution of m1()  
Thread 2 starts execution of m1()  
  
>>>>>>>>><<<<<<<<<<<<<<<  
If not synchronized...no deadlock (no locks)

# **Session 10:**

## **Learning Objectives**

By the end of this session, you must be able to

- **Collection Framework**

# Collection Framework

## Introduction:

An array is an indexed collection of fixed number of homogeneous data elements

## Limitations of Array Objects :

- 1.Arrays are fixed in size
- 2.Arrays can hold only homogeneous data elements

## Example:

```
Student[] s=new Student(1000);
s[0]=new Student();
s[1]=new Student();
s[2]=new Customer(); // CE – Incompatible types
```

But this problem can be resolved by using Object type arrays.

## Example:

```
Object[] a= new Object[1000];
a[0]=new Student();
a[1]=new Customer();
```

3. Arrays concept not built based on some underlying data structures

# Collection Framework

## Advantages of Collections over Arrays:

1. Collections are grow able in nature – may increase or decrease – as per requirement
2. Collections can hold both homogeneous & heterogeneous objects
3. Every collection class is implemented based on some data structures. Readymade method support is available for every requirement

## Note:

- Arrays can be used to hold both primitives & objects
- Collections can be used to hold only objects but not for primitives

## Collection:

A group of individual objects as a single entity is called “Collection”

# Collection Framework

## Collection framework:

A collections framework is a unified architecture for representing and manipulating collections

### All collections frameworks contain the following:

**Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently

**Implementations (Classes):** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.

**Algorithms (methods):** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic

## The Java Collections Framework provides the following benefits:

- ❖ Reduces programming effort
- ❖ Increases program speed and quality
- ❖ Allows interoperability among unrelated APIs
- ❖ Fosters software reuse, etc.,

# 9 – Key Interfaces of Collection Framework

## 1. Collection (Interface):

In general, **Collection** interface is considered as root interface of Collection Framework

**Collection** interface defines the most common methods which can be applicable for any collection object.

**The Collection interface contains methods that perform basic operations, such as:**

int **size()**

boolean **isEmpty()**

boolean **contains(Object element)**

boolean **add(E element)**

boolean **remove(Object element)**

iterator<E> **iterator()**

## Collection Interface Bulk Operations

Bulk operations perform an operation on an entire Collection. The following are the bulk operations:

**containsAll( )** : returns true if the target Collection contains all of the elements in the specified Collection.

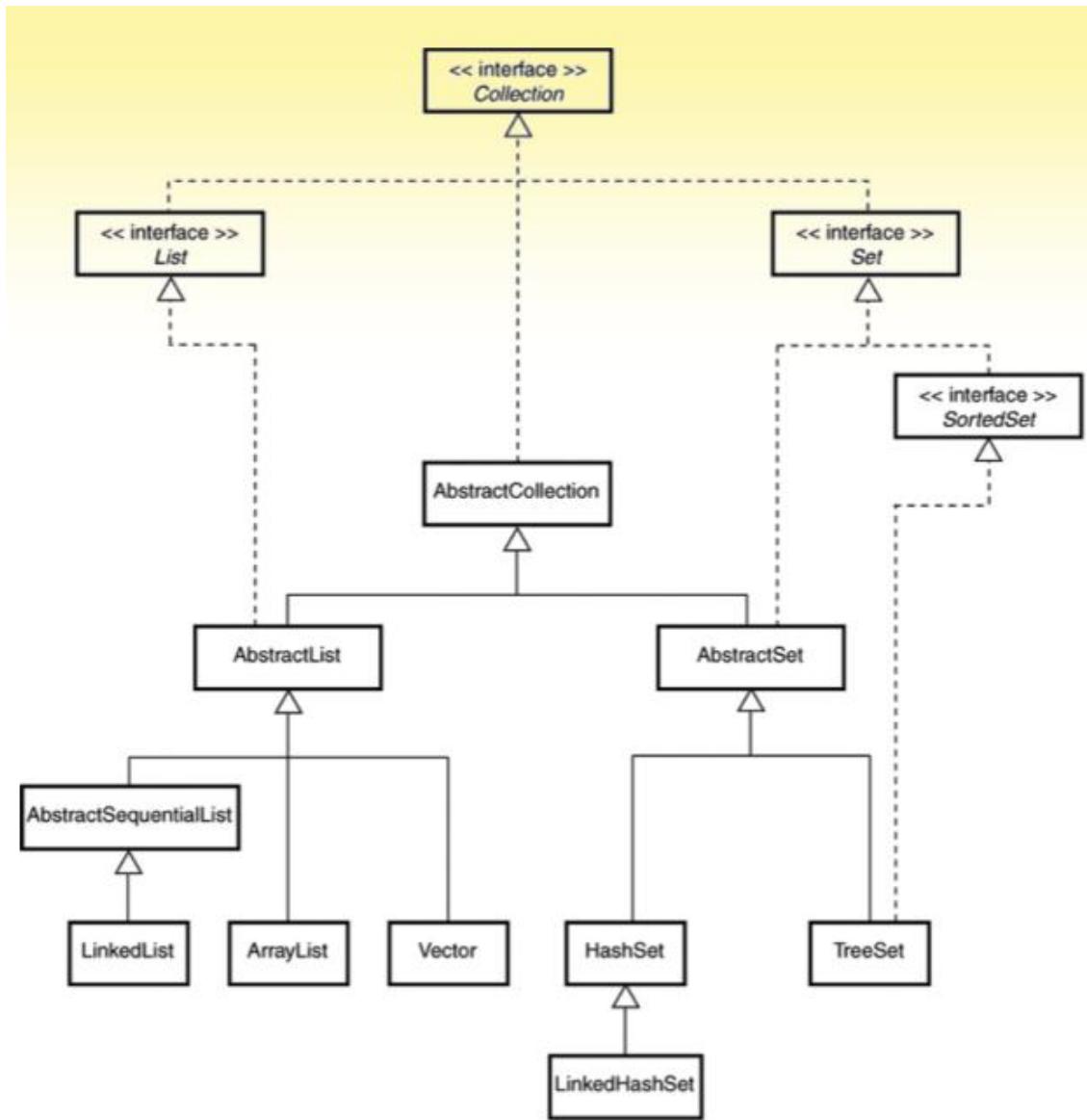
**addAll( )** : adds all of the elements in the specified Collection to the target Collection.

**removeAll( )** : removes from the target Collection all of its elements that are also contained in the specified Collection.

**retainAll( )** : it retains only those elements in the target Collection that are also contained in the specified Collection.

**clear( )** : removes all elements from the Collection.

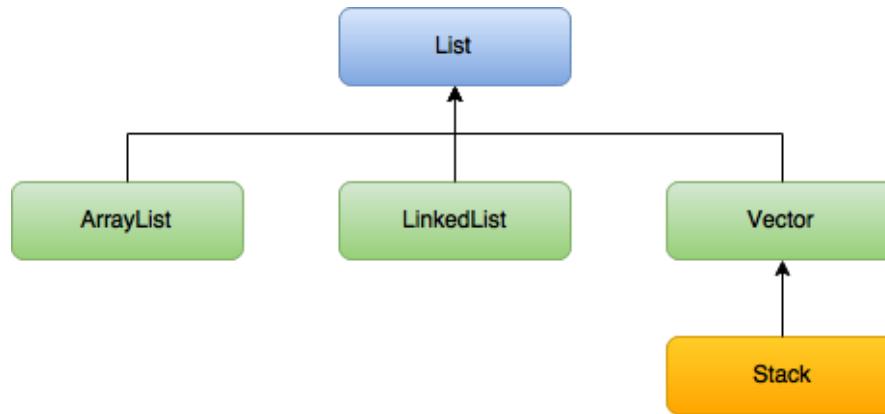
# 9 – Key Interfaces of Collection Framework



# 9 – Key Interfaces of Collection Framework

## 2. List (Interface):

- ✓ It is a child interface of Collection
- ✓ A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements.
- ✓ Used to represent a group of individual objects where **insertion order is preserved & duplicates are allowed**



\*Vector & Stack are legacy classes

In addition to the operations inherited from Collection, the List interface includes operations for the following:

- ✓ **Positional access** : manipulates elements based on their numerical position in the list.  
This includes methods such as **get()**, **set()**, **add()**, **addAll()**, and **remove()**
- ✓ **Search** : searches for a specified object in the list and returns its numerical position.  
Search methods include **indexOf()** and **lastIndexOf()**
- ✓ **Iteration** : extends Iterator semantics to take advantage of the list's sequential nature. The **listIterator** methods provide this behavior- **hasPrevious()**, **next()** and **previous()**, **hasNext()**, etc
- ✓ **Range-view** : The **subList()** method performs arbitrary range operations on the list.  
Ex: `list.subList(fromIndex, toIndex).clear(); // removes those ranged values`

# 9 – Key Interfaces of Collection Framework

**Most polymorphic algorithms in the Collections class apply specifically to List.**

**sort(list l)** : sorts a List using a merge sort algorithm, which provides a fast, stable sort.

**shuffle ( )** : randomly permutes the elements in a List.

**reverse ( )**: reverses the order of the elements in a List.

**rotate ( )**: rotates all the elements in a List by a specified distance.

**swap ( )**: swaps the elements at specified positions in a List.

**replaceAll ( )**: replaces all occurrences of one specified value with another.

**fill( )**: overwrites every element in a List with the specified value.

**copy ( )**: copies the source List into the destination List.

**binarySearch( )**: searches for an element in an ordered List using the binary search algorithm.

**indexOfSubList( )**: returns the index of the first sublist of one List that is equal to another.

**lastIndexOfSubList( )**: returns the index of the last sublist of one List that is equal to another.

# 9 – Key Interfaces of Collection Framework

## 3. Set (Interface):

- ✓ It is a child interface of Collection
- ✓ A Set is a Collection that cannot contain duplicate elements
- ✓ Used to represent a group of individual objects where **insertion order is not preserved & duplicates are not allowed**
- ✓ The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited

There are three general-purpose Set implementations:

HashSet, TreeSet, and LinkedHashSet.

### HashSet :

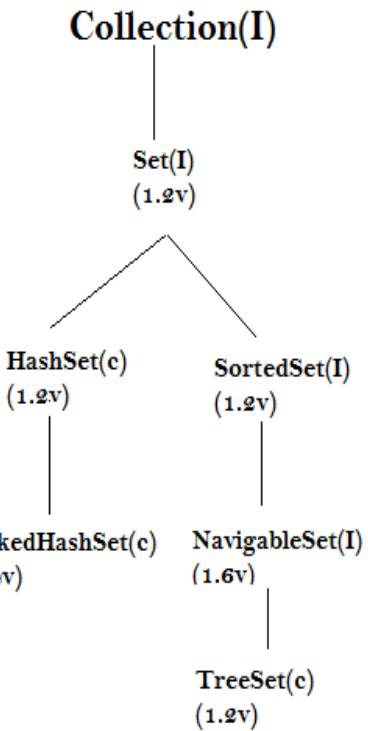
- ✓ stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration.

### TreeSet :

- ✓ stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet.

### LinkedHashSet:

- ✓ implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order).
- ✓ LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.



# 9 – Key Interfaces of Collection Framework

## Basic Operations on Set:

**size()** method returns the number of elements in the Set (its cardinality)

**isEmpty()** method returns whether the set is empty or not

**add()** method adds the specified element to the Set if it is not already present and returns a boolean indicating whether the element was added

**remove()** method removes the specified element from the Set if it is present and returns a boolean indicating whether the element was present

**iterator()** method returns an Iterator over the Set

## 4. SortedSet (Interface):

- ✓ It is a child interface of Set
- ✓ Used to represent a group of individual objects according to **some sorting order**

## 5. NavigableSet (Interface):

- ✓ It is a child interface of SortedSet
- ✓ Defines several methods for **navigation** purpose

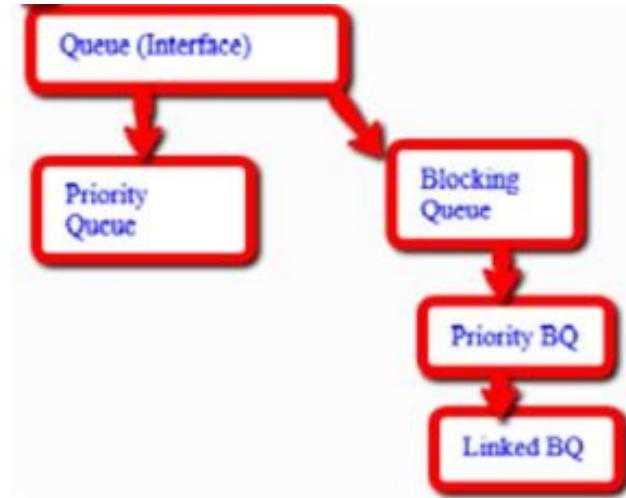
# 9 – Key Interfaces of Collection Framework

## 6. Queue (Interface):

- ✓ It is a child interface of Collection
- ✓ Used to represent a group of individual objects **prior to processing**
- ✓ A [Queue](#) is a collection for holding elements prior to processing.
- ✓ Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations.

The Queue interface follows:

- ✓ public interface Queue<E> extends Collection<E> {  
    E element(); // return head  
    boolean offer(E e); // add  
    E peek(); // return head  
    E poll(); //remove  
    E remove(); // remove  
}



### Queue Interface Structure

| Type of Operation | Throws exception | Returns special value |
|-------------------|------------------|-----------------------|
| Insert            | add(e)           | offer(e)              |
| Remove            | remove()         | poll()                |
| Examine           | element()        | peek()                |

# 9 – Key Interfaces of Collection Framework

## Deque (Interface):

- ✓ Usually pronounced as deck, a **deque** is a double-ended-queue.
- ✓ A double-ended-queue is a linear collection of elements that supports the **insertion** and **removal** of elements at **both end points**.
- ✓ The **Deque** interface, defines methods to access the elements at both ends of the Deque instance.
- ✓ Methods are provided to **insert, remove, and examine** the elements.

| Deque Methods     |                                                 |                                          |
|-------------------|-------------------------------------------------|------------------------------------------|
| Type of Operation | First Element (Beginning of the Deque instance) | Last Element (End of the Deque instance) |
| Insert            | addFirst(e)<br>offerFirst(e)                    | addLast(e)<br>offerLast(e)               |
| Remove            | removeFirst()<br>pollFirst()                    | removeLast()<br>pollLast()               |
| Examine           | getFirst()<br>peekFirst()                       | getLast()<br>peekLast()                  |

- ❖ All the above interfaces (**Collection, List, Set, SortedSet, NavigableSet, Queue**) used to represent a group of individual objects only.
- ❖ To represent group of objects as key-value pairs , then **Map** interface has to be used

# 9 – Key Interfaces of Collection Framework

## 7. Map (Interface):

- ✓ A Map is an object that maps keys to values.
- ✓ A map cannot contain duplicate keys: Each key can map to at most one value.
- ✓ Map is used to represent a group of individual objects as **key-value pairs** (Ex: id - name)
- ✓ Both key and value are objects only
- ✓ Duplicate keys are not allowed, but values can be duplicated
  
- ✓ The Map interface includes methods for **basic operations**: put(), get(), remove(), containsKey(), containsValue(), size(), and empty()
  
- ✓ **Bulk operations**: (putAll() and clear()), and collection views (such as keySet(), entrySet(), and values()).

### Collection Views

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

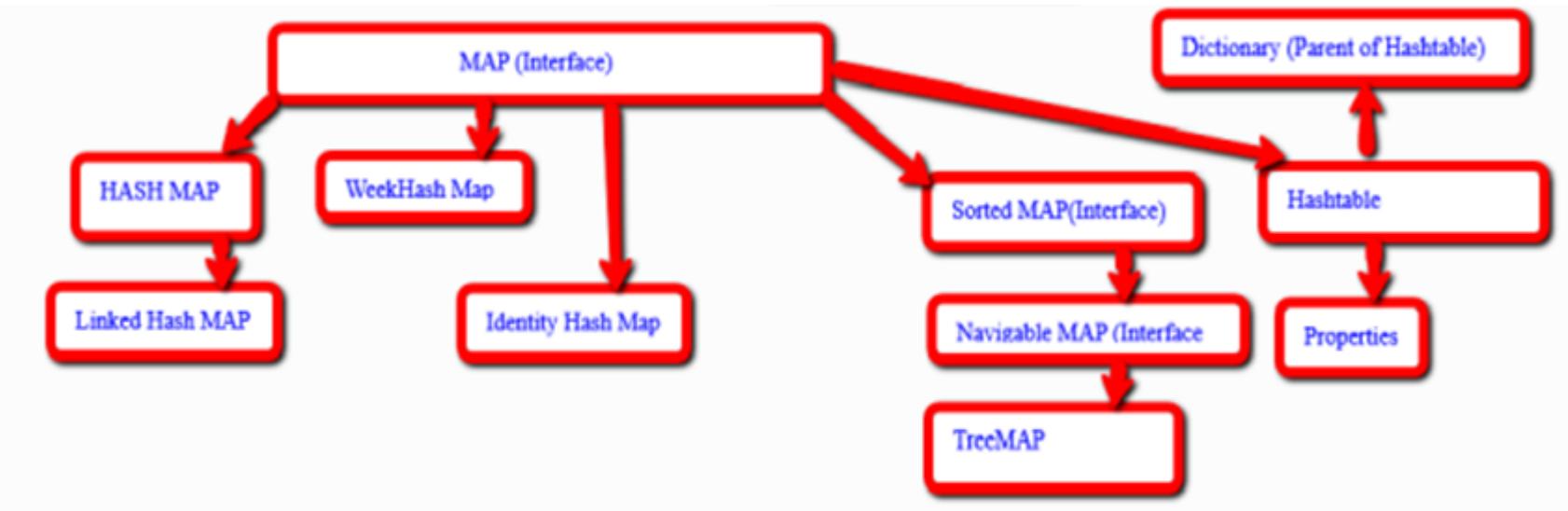
**keySet** — the Set of keys contained in the Map.

**values** — The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.

**entrySet** — the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called **Map.Entry**, the type of the elements in this Set.

# 9 – Key Interfaces of Collection Framework

## 7. Map (Interface):



- ❖ Map is not child interface of Collection
- ❖ Hashtable, Properties and Dictionary are the legacy classes

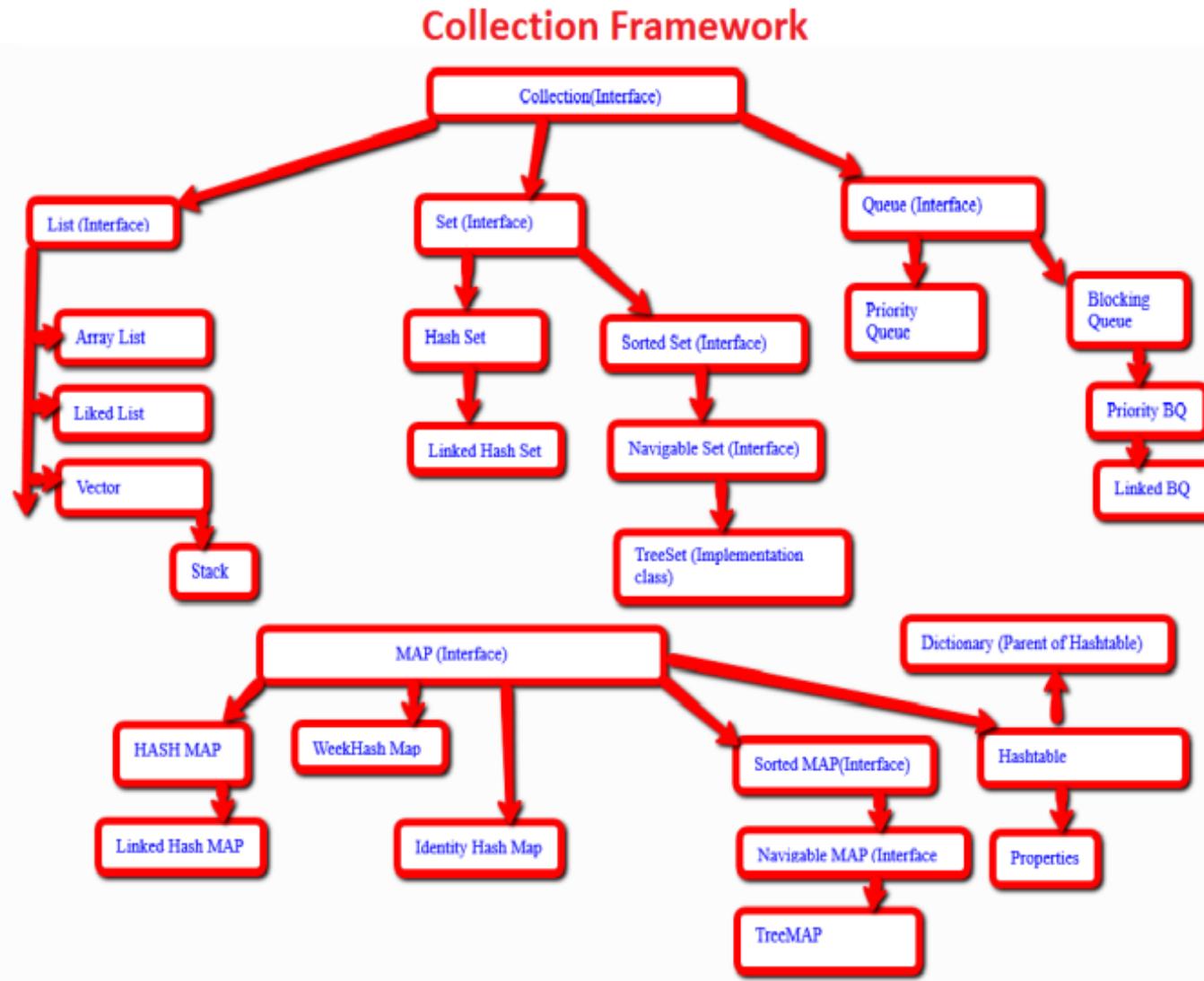
## 8. SortedMap (Interface):

- ✓ It's a child interface of Map
- ✓ Used to represent a group of individual objects as **key-value** pairs according to some **sorting order**
- ✓ Sorting should be done only based on **keys** but not on values

## 9. NavigableMap (Interface):

- ✓ It's a child interface of SortedMap
- ✓ Defines several methods for **navigation** purpose

# Collection Framework - Summary



# Collection Framework : and more..

## Utility Classes:

1. Arrays - *applies for arrays*
2. Collections - *A List 'l' may be sorted as follows:*  
*Collections. Sort(l);*

## Cursors (Iterators):

1. Enumeration - *for legacy classes*
2. Iterator - *Universal iterator*
3. ListIterator – *list types*

## Interfaces (for Sorting):

1. Comparable – default order – natural sorting prder
2. Comparator - any other order

## Array List Demo

```
import java.util.ArrayList;
import java.util.*;
public class ALDemo {
 public static void main(String[] args) {
 ArrayList al=new ArrayList();
 int[] a={5,7,9,2,3,1};
 Arrays.sort(a);
 for(int i:a)
 System.out.println(i);

 al.add("sreenivas");
 al.add(10);
 al.add(101);
 al.add(3.14);
 al.add('s');
 al.add(true);
 ArrayList al1=new ArrayList();
 al1.add("vasu");
 al1.add("sreenivas");
 al.addAll(al);
 al.add(1,"Sadhu");
 al.add(0,"cdac");
 Collections.sort(al1);
 System.out.println(al);

 Iterator itr=al.iterator();
 while(itr.hasNext()){
 System.out.println(itr.next());
 }
 }
}
```

# Array List Demo

```
import java.util.*;

public class ALDemo1 {
 public static void main(String[]
args) {

 ArrayList al=new ArrayList();
 al.add("Sadhu");
 al.add("Sreenivas");
 al.add("35");
 al.add("8.5");
 al.add("Hyderabad");
 al.add("500089");
 al.add("true");
 System.out.println(al);

 // Collections.sort(al);
 System.out.println(al);

 Iterator itr=al.iterator();
 while(itr.hasNext())
 System.out.println(itr.next());

 }
}
```

## Linked List Demo

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.ListIterator;

public class LLDemo1 {

 public static void main(String[] args) {
 LinkedList al=new LinkedList();
 al.add("Sadhu");
 al.add("Sreenivas");
 al.add("35");
 al.add("8.5");
 al.add("Hyderabad");
 al.add("500089");
 al.add("true");
 al.addFirst("Mr");
 al.addLast("false");
 System.out.println(al);
 // Collections.sort(al);
 System.out.println(al);
 ListIterator itr=al.listIterator();
 while(itr.hasNext())
 {
 System.out.println(itr.next());
 }
 System.out.println(itr.previous());
 }
}
```

## Stack Demo

```
import java.util.*;
public class StackDemo {
 public static void main(String[] args) {

 Stack s=new Stack();
 s.push(10);
 s.push(20);
 s.push(30);
 s.push(40);
 s.push(50);
 System.out.println(s);
 System.out.println(s.peek());
 System.out.println(s.pop());
 System.out.println(s.pop());
 System.out.println(s.peek());
 }
}
```

# Vector Demo

```
import java.util.*;

public class VectorDemo {

 public static void main(String[] args) {

 Vector v=new Vector();
 System.out.println(v.capacity());
 v.add("Sadhu");
 v.addElement("Sreeni");
 v.add("Hyderabad");
 System.out.println(v);

 v.remove(0);
 Collections.sort(v);
 Enumeration e=v.elements();
 while(e.hasMoreElements())

 System.out.println(e.nextElement());

 Iterator itr=v.iterator();
 while(itr.hasNext())
 System.out.println(itr.next());
 }
}
```

## HashSet Demo

```
import java.util.HashSet;

import java.util.*;
public class HashSetDemo {

 public static void main(String[] args) {

 LinkedHashSet h=new LinkedHashSet();
 h.add("TS");
 h.add("AP");
 h.add("UP");
 h.add("TS"); // false
 h.add(123);
 h.add(321);
 h.add(null);
 System.out.println(h);

 Iterator i=h.iterator();
 while(i.hasNext()){
 System.out.println(i.next());
 }
 }
}
```

## TreeSet Demo

```
import java.util.*;
public class TreeSetDemo {

 public static void main(String[] args) {

 TreeSet t=new TreeSet();
 t.add(10);
 t.add(9);
 t.add(11);
 t.add(5);
 t.add(15);
 //t.add(null); not possible
 // t.add("ABC");
 System.out.println(t);
 System.out.println(t.first());
 System.out.println(t.last());
 System.out.println(t.headSet(11));
 System.out.println(t.tailSet(10));

 Iterator itr=t.iterator();
 while(itr.hasNext())
 System.out.println(itr.next());
 }
}
```

## PriorityQueue Demo

```
import java.util.*;
public class PriorityQueueDemo {
 public static void main(String[] args) {

 PriorityQueue pq=new
PriorityQueue();
 pq.add(10);
 pq.add(20);
 pq.add(500);
 pq.add(5);
 pq.add(50);
 pq.add(200);
 pq.offer(1000);
 System.out.println(pq);
 pq.remove();
 pq.remove();
 pq.remove();
 System.out.println(pq.peek());
 System.out.println(pq);
 }
}
```

## ListIterator Demo

```
import java.io.*;
import java.util.*;

public class ListIteratorDemo {

 public static void main(String[] args) {
 LinkedList l=new LinkedList();
 l.add("sachin");
 l.add("saurabh");
 l.add("yuvi");
 l.add("dhoni");
 l.add("zaheer");
 System.out.println(l);

 ListIterator litr=l.listIterator();
 while(litr.hasNext()){
 String s=(String)litr.next();
 if(s.equals("zaheer"))
 litr.remove();
 if(s.equals("sachin"))
 litr.set("Virat");

 }
 System.out.println(l);
 }
}
```

## HashMap Demo

```
import java.util.*;

public class HashMapDemo {
 public static void main(String[] args) {
 HashMap m=new HashMap();
 m.put("sadhu",5400);
 m.put("simi",5400);
 m.put("sharan",6600);
 m.put("pramod",7600);
 System.out.println(m);
 m.put("nag",5400);
 m.put("bsrk",5400);
 System.out.println(m);

 //Set s=m.keySet();
 //System.out.println(s);

 Collection s1=m.keySet();
 System.out.println(s1);
 Collection c=m.values();
 System.out.println(c);

 Set s2=m.entrySet();
 Iterator itr=s2.iterator();
 while(itr.hasNext()){
 Map.Entry m1=(Map.Entry)itr.next();
 System.out.println(m1.getKey()+" __ "+m1.getValue());
 }
 }
}
```

## TreeMap Demo

```
import java.util.*;

public class TreeMapDemo {
 public static void main(String[] args) {

 TreeMap m=new TreeMap(new MyComparator());
 m.put(105,"asdf");
 m.put(101, "pqr");
 m.put(102, "abc");
 m.put(103, "xyz");
 m.put(104, "mno");
 System.out.println(m);
 }
}
class MyComparator implements Comparator{
 public int compare(Object o1, Object o2){
 String s1=o1.toString();
 String s2=o2.toString();
 return s2.compareTo(s1);
 }
}
```

## IdentityHashMap Demo

```
import java.util.*;
public class IdentityHashMapDemo {

 public static void main(String[] args) {

 IdentityHashMap m=new IdentityHashMap();
 Integer i1=new Integer(10);
 Integer i2=new Integer(10);
 m.put(i1,"sadhu");
 m.put(i2,"sreeni");
 System.out.println(m);
 }
}
```

## Summary of Interfaces

➤ The core collection interfaces are the foundation of the Java Collections Framework.

**The Java Collections Framework hierarchy consists of two distinct interface trees:**

- **The first tree** starts with the **Collection** interface, which provides for the basic functionality used by all collections, such as add and remove methods.
- Its sub interfaces — Set, List, and Queue — provide for more specialized collections.
- The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface, SortedSet, that provides for ordering of elements in the set.
- The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
- The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in on a FIFO basis.
- The Deque interface enables insertion, deletion, and inspection operations at both the ends. Elements in a Deque can be used in both LIFO and FIFO.

**The second tree** starts with the **Map** interface, which maps keys and values similar to a Hashtable.

Map's subinterface, SortedMap, maintains its key-value pairs in ascending order or in an order specified by a Comparator.

These interfaces allow collections to be manipulated independently of the details of their representation.

## Commonly used methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

| Method                                          | Description                                                                                |
|-------------------------------------------------|--------------------------------------------------------------------------------------------|
| <b>public boolean add(object element)</b>       | is used to insert an element in this collection.                                           |
| <b>public boolean addAll(collection c)</b>      | is used to insert the specified collection elements in the invoking collection.            |
| <b>public boolean remove(object element)</b>    | is used to delete an element from this collection.                                         |
| <b>public boolean removeAll(Collection c)</b>   | is used to delete all the elements of specified collection from the invoking collection.   |
| <b>public boolean retainAll(Collection c)</b>   | is used to delete all the elements of invoking collection except the specified collection. |
| <b>public int size()</b>                        | return the total number of elements in the collection.                                     |
| <b>public void clear()</b>                      | removes the total no of element from the collection.                                       |
| <b>public boolean contains(object element)</b>  | is used to search an element.                                                              |
| <b>public boolean containsAll(collection c)</b> | is used to search the specified collection in this collection.                             |
| <b>public Iterator iterator()</b>               | returns an iterator.                                                                       |

## Iterator interface

Iterator interface provides the facility of iterating the elements in forward direction only.

### Methods of Iterator interface

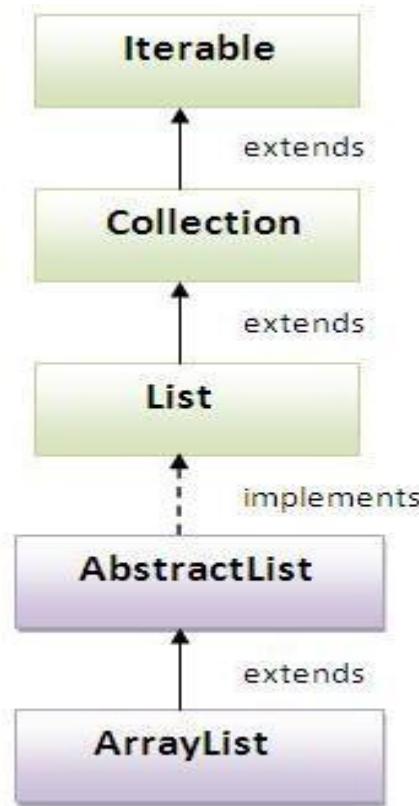
There are only three methods in the Iterator interface. They are:

1. **public boolean hasNext()** it returns true if iterator has more elements.
2. **public object next()** it returns the element and moves the cursor pointer to the next element.
3. **public void remove()** it removes the last elements returned by the iterator. It is rarely used.

## ArrayList class:

- Uses a dynamic array for storing the elements. It extends AbstractList class and implements List interface.
- Can contain duplicate elements.
- Maintains insertion order.
- Not synchronized.
- Random access because array works at the index basis.
- Manipulation slow because a lot of shifting needs to be occurred.

### Hierarchy of ArrayList class:



## Example of ArrayList:

```
import java.util.*;
class Simple{
public static void main(String args[]){

 ArrayList al=new ArrayList();
 al.add("Ravi");
 al.add("Vijay");
 al.add("Ravi");
 al.add("Ajay");

 Iterator itr=al.iterator();
 while(itr.hasNext()){
 System.out.println(itr.next());
 }
}
}
```

**Output :**  
Ravi  
Vijay  
Ravi  
Ajay

## Storing user-defined class objects:

```
class Student{
 int rollno;
 String name;
 int age;
 Student(int rollno, String name, int age){
 this.rollno=rollno;
 this.name=name;
 this.age=age;
 }
}
```

```
import java.util.*;
class Simple{
 public static void main(String args[]){

 Student s1=new Student(101,"Sonoo",23);
 Student s2=new Student(102,"Ravi",21);
 Student s3=new Student(103,"Hanumat",25);

 ArrayList al=new ArrayList();
 al.add(s1);
 al.add(s2);
 al.add(s3);

 Iterator itr=al.iterator();
 while(itr.hasNext()){
 Student st=(Student)itr.next();
 System.out.println(st.rollno+" "+st.name+" "+st.age);
 }
 }
}
```

**Output:**  
101 Sonoo 23  
102 Ravi 21  
103 Hanumat 25

## Two ways to iterate the elements of collection:

1. By Iterator interface.
2. By for-each loop.

### Iterating the elements of Collection by for-each loop:

```
import java.util.*;
class Simple{
 public static void main(String args[]){

 ArrayList al=new ArrayList();
 al.add("Ravi");
 al.add("Vijay");
 al.add("Ravi");
 al.add("Ajay");

 for(Object obj:al)
 System.out.println(obj);
 }
}
```

**Output:**  
Ravi  
Vijay  
Ravi  
Ajay

## Example of addAll(Collection c) method:

```
import java.util.*;
class Simple{
 public static void main(String args[]){

 ArrayList al=new ArrayList();
 al.add("Ravi");
 al.add("Vijay");
 al.add("Ajay");

 ArrayList al2=new ArrayList();
 al2.add("Sonoo");
 al2.add("Hanumat");

 al.addAll(al2);

 Iterator itr=al.iterator();
 while(itr.hasNext()){
 System.out.println(itr.next());
 }
 }
}
```

---

**Output :**

```
Ravi
Vijay
Ajay
Sonoo
Hanumat
```

## Example of removeAll() method:

```
import java.util.*;
class Simple{
 public static void main(String args[]){

 ArrayList al=new ArrayList();
 al.add("Ravi");
 al.add("Vijay");
 al.add("Ajay");

 ArrayList al2=new ArrayList();
 al2.add("Ravi");
 al2.add("Hanumat");

 al.removeAll(al2);

 System.out.println("iterating the elements after removing the elements of al2...");
 Iterator itr=al.iterator();
 while(itr.hasNext()){
 System.out.println(itr.next());
 }
 }
}
```

**Output:** iterating the elements after removing the elements of al2...  
Vijay  
Ajay

## Example of retainAll() method:

```
import java.util.*;
class Simple{
 public static void main(String args[]){

 ArrayList al=new ArrayList();
 al.add("Ravi");
 al.add("Vijay");
 al.add("Ajay");

 ArrayList al2=new ArrayList();
 al2.add("Ravi");
 al2.add("Hanumat");

 al.retainAll(al2);

 System.out.println("iterating the elements after retaining the elements of al2...");
 Iterator itr=al.iterator();
 while(itr.hasNext()){
 System.out.println(itr.next());
 }
 }
}
```

**Output:** iterating the elements after retaining the elements of al2...  
Ravi

# Generics

**Java Generics** programming is introduced in J2SE 5 to deal with type-safety of objects

Earlier to Generics, used to store any type of objects in collection.

Now, in generics, java programmer is forced to store specific type of objects

## **Advantage of Java Generics:**

### **1)Type-safety :**

Holds only a single type of objects in generics. It doesn't allow to store other typed objects

### **2)Type casting is not required:**

There is no need to typecast the object

### **3)Compile-Time Checking:**

It is checked at compile time but not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

# Generics

**Earlier to Generics, type cast is used.**

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

**After Generics, don't need to typecast the object.**

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

**Compile Time Checking:**

```
List<String> list = new ArrayList<String>();
list.add("hello");
list.add(32); //Compile Time Error
```

# Generics

## Example of Generics in Java

We have just seen ArrayList class, also can use any collection class such as LinkedList, HashSet, TreeSet, HashMap, Comparator etc.

```
import java.util.*;

class TestGenerics1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();
list.add("ABC");
list.add("XYZ");
//list.add(32);//compile time error
```

```
String s=list.get(1);//type casting is not required
System.out.println("element is: "+s);
```

```
Iterator<String> itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
```

# Generics

## Example of Java Generics using Map

```
import java.util.*;
class TestGenerics2{

public static void main(String args[]){

Map<Integer,String> map=new HashMap<Integer,String>();
map.put(1,"ABC");
map.put(4,"XYZ");
map.put(2,"PQR");

//Now use Map.Entry for Set and Iterator
Set<Map.Entry<Integer,String>> set=map.entrySet();

Iterator<Map.Entry<Integer,String>> itr=set.iterator();
while(itr.hasNext()){
Map.Entry e=itr.next(); //no need to typecast
System.out.println(e.getKey()+" "+e.getValue());
}

}
```

# Generics

## Generic class

A class that can refer to any type is known as generic class.

Here, we are using **T** type parameter to create the generic class of specific type.

**Creating generic class:**

```
class MyGen<T>{
 T obj;
 void add(T obj){
 this.obj=obj;
 }
 T get(){
 return obj;
 }
}
```

The **T** type indicates that it can refer to any type (like String, Integer, Employee etc.). The type you specify for the class, will be used to store and retrieve the data.

```
class TestGenerics3{
 public static void main(String args[]){
 MyGen<Integer> m=new MyGen<Integer>();
 m.add(2);
 //m.add("ABC");//Compile time error
 System.out.println(m.get());
 }
}
```

## Generic Class - Example

```
class MyGen<T>{
 T obj;
 void add(T obj){
 this.obj=obj;
 }
 T get(){
 return obj;
 }
}
public class GenericDemo {
 public static void main(String[] args) {
 MyGen<Integer> m1=new MyGen();
 m1.add(99);
 System.out.println(m1.get());
 // m1.add("AXBC");
 MyGen<String> m2=new MyGen();
 m2.add("Hello");
 System.out.println(m2.get());
 }
}
```

# Generics

## Type Parameters

The type parameters naming conventions are important to learn generics thoroughly.

The commonly type parameters are as follows:

T - Type

E - Element

K - Key

N - Number

V - Value

## Generic Method

Like generic class, we can create generic method that can accept any type of argument. **E** to denote the element.

```
public class TestGenerics4{
 public static < E > void printArray(E[] elements) {
 for (E element : elements){
 System.out.println(element);
 }
 System.out.println();
 }
 public static void main(String args[]) {
 Integer[] intArray = { 10, 20, 30, 40, 50 };
 Character[] charArray = { 'A', 'B', 'C', 'D', 'E','F','G','H','I','J' };

 System.out.println("Printing Integer Array");
 printArray(intArray);

 System.out.println("Printing Character Array");
 printArray(charArray);
 }
}
```

# Generics - wildcard

```
import java.util.*;
public class GenericWildCard {
 public static void main(String[] args) {
 List<Integer> ints = new ArrayList<>();
 ints.add(3); ints.add(5); ints.add(10);
 double sum = sum(ints); // Incompatible types List<Integer> can not be converted to List<Number>
 System.out.println("Sum of ints="+sum);
 }

 public static double sum(List<Number> list){
 double sum = 0;
 for(Number n : list){
 sum += n.doubleValue();
 }
 return sum;
 }
}
```

- ❑ Now the problem with above implementation is that it won't work with List of Integers or Doubles because we know that List<Integer> and List<Double> are not related, this is when upper bounded wildcard is helpful.
- ❑ We use generics wildcard with extends keyword and the upper bound class or interface that will allow us to pass argument of upper bound or its subclasses types.

# Generics - wildcard

```
import java.util.*;
public class GenericWildCard {
 public static void main(String[] args) {
 List<Integer> ints = new ArrayList<>();
 ints.add(3); ints.add(5); ints.add(10);
 double sum = sum(ints);
 System.out.println("Sum of ints="+sum);
 }

 public static double sum(List<? extends Number> list){
 double sum = 0;
 for(Number n : list){
 sum += n.doubleValue();
 }
 return sum;
 }
}
```

# Generics - Wildcard

The ? (question mark) symbol represents wildcard element. It means any type. If we write <? extends Number>, it means any child class of Number e.g. Integer, Float, Double etc. Now, we can call the method of Number class through any child class object.

```
import java.util.*;

abstract class Shape{
 abstract void draw();
}

class Rectangle extends Shape{
 void draw(){
 System.out.println("drawing rectangle");
 }
}

class Circle extends Shape{
 void draw(){
 System.out.println("drawing circle");
 }
}
```

```
class GenericTest {
 //creating a method that accepts only child class of Shape

 public static void drawShapes(List<? extends Shape> lists){
 for(Shape s:lists){
 s.draw(); //calling method of Shape class by child class instance
 }
 }
 public static void main(String args[]){
 List<Rectangle> list1=new ArrayList<Rectangle>();
 list1.add(new Rectangle());

 List<Circle> list2=new ArrayList<Circle>();
 list2.add(new Circle());
 list2.add(new Circle());

 drawShapes(list1);
 drawShapes(list2);
 }
}
// drawing a rectangle!
//drawing a circle!
//drawing a circle!
```

# Java Generics Unbounded Wildcard

Sometimes we have a situation where we want our generic method to be working with all types, in this case unbounded wildcard can be used. Its same as using <? extends Object>.

```
import java.util.*;
public class GenericWildCard {
 public static void main(String[] args) {
 List<Integer> l1 = new ArrayList<>();
 l1.add(3); l1.add(5); l1.add(10);
 printData(l1);

 List<String> l2=new ArrayList<>();
 l2.add("ABC");
 l2.add("XYZ");
 printData(l2);
 }

 public static void printData(List<?> list){
 for(Object obj : list){
 System.out.print(obj + " ");
 }
 }
}
```

# Java Generics Lower bounded Wildcard

Suppose we want to add Integers to a list of integers in a method, we can keep the argument type as `List<Integer>` but it will be tied up with Integers whereas `List<Number>` and `List<Object>` can also hold integers, so we can use lower bound wildcard to achieve this.

We use generics wildcard (?) with **super** keyword and lower bound class to achieve this.

We can pass lower bound or any super type of lower bound as an argument in this case, java compiler allows to add lower bound object types to the list.

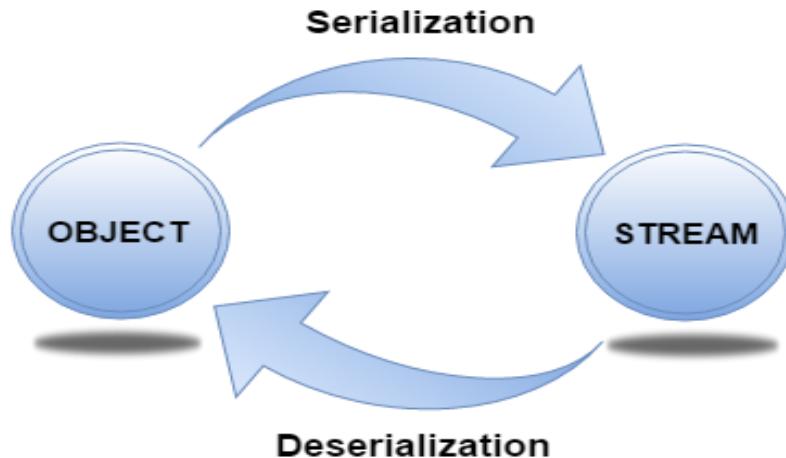
```
public static void addIntegers(List<? super Integer> list){
 list.add(new Integer(50));
}
```

# Serialization

**Serialization in java** is a mechanism of *writing the state of an object into a byte stream*.

The reverse operation of serialization is called *deserialization*.

It is mainly used to pass/share object's state on the network (known as marshalling).



**Java transient keyword** is used in serialization. If you define any data member as transient, it will not be serialized.

# Serialization

```
import java.io.*;
class Person implements Serializable //marker interface{
transient int age=30;
String name="ABC";
}
class SerializeTest{
static public void main(String[] args) throws Exception{
Person p1=new Person();
//serialization
FileOutputStream fos=new FileOutputStream("abc.ser");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(p1);

//Deserialization
FileInputStream fis=new FileInputStream("abc.ser");
ObjectInputStream ois=new ObjectInputStream(fis);
Person p2=(Person)ois.readObject();
System.out.println(p1.age+" "+p1.name+" "+p2.age+" "+p2.name);
}
} //output: 30 ABC 0 ABC transient means – value will not be serialized
```

# Serialization

```
import java.io.*;

class Person implements Serializable{
transient static int age=30;
String name="ABC";
}
class SerializeTest{
static public void main(String[] args) throws Exception{
Person p1=new Person();
//serialization
FileOutputStream fos=new FileOutputStream("abc.txt");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(p1);
//Deserialization
FileInputStream fis=new FileInputStream("abc.txt");
ObjectInputStream ois=new ObjectInputStream(fis);
Person p2=(Person)ois.readObject();
System.out.println(p1.age+" "+p1.name+" "+p2.age+" "+p2.name);
}
} //30 ABC 30 ABC
```

# Serialization

```
import java.io.*;

class Person implements Serializable{
transient static int age=30;
String name="ABC";
}
class SerializeTest{
static public void main(String[] args) throws Exception{
Person p1=new Person();
//serialization
FileOutputStream fos=new FileOutputStream("abc.txt");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(p1);
//Deserialization
FileInputStream fis=new FileInputStream("abc.txt");
ObjectInputStream ois=new ObjectInputStream(fis);
Person p2=(Person)ois.readObject();
System.out.println(p1.age+" "+p1.name+" "+p2.age+" "+p2.name);
}
} //30 ABC 30 ABC
```

# Serialization

```
import java.io.*;
class Person implements Serializable{
transient final int age=30;
transient String name="ABC";
}
class SerializeTest{
static public void main(String[] args) throws Exception{
Person p1=new Person();
//serialization
FileOutputStream fos=new FileOutputStream("abc.txt");
ObjectOutputStream oos=new ObjectOutputStream(fos);
oos.writeObject(p1);

//Deserialization
FileInputStream fis=new FileInputStream("abc.txt");
ObjectInputStream ois=new ObjectInputStream(fis);
Person p2=(Person)ois.readObject();

System.out.println(p1.age+" "+p1.name+" "+p2.age+" "+p2.name);
}
} // output: 30 ABC 30 null
```

# Reflection API

- **Reflection** is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine
- The ability of a computer program to examine and modify the structure and behavior of program at run time
- This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language
- With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible

## Reflection API is used to

- Inspect class and method modifiers
    - private, public, final and abstract
  - Inspect constructors, methods and their parameters
  - Get and Set private data
  - Invoke public / private methods
- **java.lang.reflect.\*** - need to be imported for reflection API

# Reflection API

- Every object is either a reference or primitive type
- Reference types all inherit from [java.lang.Object](#)
- Classes, enums, arrays, and interfaces are all reference types.
- Examples of reference types include [java.lang.String](#), [java.lang.Double](#), [java.io.Serializable](#), [javax.swing.SortOrder](#)
- There is a fixed set of primitive types: boolean, byte, short, int, long, char, float, and double.
- ***For every type of object, the Java Virtual Machine instantiates an immutable instance of [java.lang.Class](#) which provides methods to examine the runtime properties of the object including its members and type information.***
- [Class](#) also provides the ability to create new classes and objects.
- Most importantly, it is **the entry point** for all of the Reflection APIs.

# Reflection API

**Simple example to read methods and their parameter types**

```
import java.lang.reflect.Method;

public class ReflectionDemo {
 public static void main(String[] args) {
 Class c="foo".getClass();
 System.out.println(c.getName());
 Method[] strMethods=c.getDeclaredMethods();
 for(Method m:strMethods) {
 System.out.println("Look at method:"+m.getName());
 Class<?> parameterType[]=m.getParameterTypes();
 for(int i=0;i<parameterType.length;i++)
 System.out.println("Parameter "+(i+1)+" parameter type :" +parameterType[i].getName());
 }
 }
}
```

# Class "Object" : getClass()

```
import java.lang.reflect.Field;
import java.lang.reflect.Method;

public class ObjectTest {
 public static void main(String[] args) {
 int count=0;
 Object o=new String("CDAC Hyderabad");
 Class c=o.getClass();
 System.out.println("FQN of class:"+c.getName());
 Method[] m=c.getDeclaredMethods(); //reflection
 Field[] f=c.getDeclaredFields(); // reflection
 for(Method m1:m){
 count++;
 System.out.println(m1.getName());
 }
 System.out.println("No of methods:"+count);
 System.out.println(".....");
 for(Field f1:f){
 count++;
 System.out.println(f1.getName());
 }
 }
}
```

## Output:

|                        |                        |
|------------------------|------------------------|
| FQN of                 | getBytes               |
| class:java.lang.String | getBytes               |
| equals                 | getBytes               |
| toString               | getBytes               |
| hashCode               | getChars               |
| compareTo              | getChars               |
| compareTo              | indexOfSupplementary   |
| indexOf                | intern                 |
| indexOf                | isEmpty                |
| indexOf                | join                   |
| indexOf                | join                   |
| indexOf                | lastIndexOf            |
| valueOf                | length                 |
| valueOf                | matches                |
| valueOf                | nonSyncContentEquals   |
| valueOf                | offsetByCodePoints     |
| valueOf                | regionMatches          |
| valueOf                | regionMatches          |
| valueOf                | replace                |
| valueOf                | replace                |
| valueOf                | replaceAll             |
| valueOf                | replaceFirst           |
| charAt                 | split                  |
| checkBounds            | split                  |
| codePointAt            | startsWith             |
| codePointBefore        | startsWith             |
| codePointCount         | subSequence            |
| compareIgnoreCase      | substring              |
| concat                 | toCharArray            |
| contains               | toLowerCase            |
| contentEquals          | toLowerCase            |
| contentEquals          | toUpperCase            |
| copyValueOf            | toUpperCase            |
| copyValueOf            | trim                   |
| endsWith               | No of methods:77       |
| equalsIgnoreCase       | .....                  |
| format                 | value                  |
| format                 | hash                   |
|                        | serialVersionUID       |
|                        | serialPersistentFields |
|                        | CASE_INSENSITIVE_ORDER |

# Reflection API

## Private Data

```
public final class TestClass {
 private int tid=55;
 private String tstr="This is confidential! ";

 public int getId(){
 return tid;
 }
 private String getStr(String str){
 return str;
 }

 public static void main(String[] args) {
 TestClass tc=new TestClass();
 Class c1=tc.getClass();
 Field[] fields=c1.getDeclaredFields();
 for(Field f:fields)
 System.out.println(f.getName()+" "+Modifier.toString(f.getModifiers()));
 }
}
o/p:
tid private
tstr private
```

# Reflection API

## Accessing Private Data

```
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;

public final class TestClass {
 private int tid=55;
 private String tstr="This is confidential! ";
 public int getId(){
 return tid;
 }
 private String getStr(String str){
 return str;
 }
 public static void main(String[] args) throws Exception{
 TestClass tc=new TestClass();
 Class c1=tc.getClass();
 Field[] fields=c1.getDeclaredFields();
 for(Field f:fields)
 System.out.println(f.getName()+" "+Modifier.toString(f.getModifiers()));

 Field str=c1.getDeclaredField("tstr");
 str.setAccessible(true);
 String whatsintstr=(String)str.get(tc);
 System.out.println("Information hiding in tstr is:"+whatsintstr);
 }
}
```

\* Private methods too can be invoked in the same way!!

# Annotations

- *Annotations*, a form of metadata, provide data about a program that is not part of the program itself.
- Annotations have no direct effect on the operation of the code they annotate.
- **Annotations have a number of uses :**
  - ✓ **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings.
  - ✓ **Compile-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
  - ✓ **Runtime processing** — Some annotations are available to be examined at runtime.
- In its simplest form, an annotation looks like : @Entity
- The at sign character (@) indicates to the compiler that what follows is an annotation.

## Predefined Annotation Types

- A set of annotation types are predefined in the Java SE API.
- Some annotation types are used by the Java compiler, and some apply to other annotations.
- The predefined annotation types defined in java.lang are
  - @Deprecated
  - @Override
  - @SuppressWarnings.

# Annotations

- **@Deprecated** annotation indicates that the marked element is *deprecated* and should no longer be used.
- The compiler generates a warning whenever a program uses a method, class, or field with the @Deprecated annotation
- Example:

```
@Deprecated static void deprecatedMethod() {
 //code goes here...
}
```

- **@Override** annotation informs the compiler that the element is meant to override an element declared in a superclass.
- Example:

```
@Override int overriddenMethod() {
}
```

- While it is not required to use this annotation when overriding a method, it helps to prevent errors.
- If a method marked with @Override fails to correctly override a method in one of its super classes, the compiler generates an error.

# Annotations

- **@SuppressWarnings** annotation tells the compiler to suppress specific warnings that it would otherwise generate.
- In the following example, a deprecated method is used, and the compiler usually generates a warning. In this case, however, the annotation causes the warning to be suppressed.

```
// use a deprecated method and tell
// compiler not to generate a warning
@SuppressWarnings("deprecation") void useDeprecatedMethod() {
 // deprecation warning
 // - suppressed objectOne.deprecatedMethod(); }
```

- Every compiler warning belongs to a category. The Java Language Specification lists two categories: **deprecation** and **unchecked**.
- The unchecked warning can occur when interfacing with legacy code written before the advent of [generics](#).
- To suppress multiple categories of warnings, use the following syntax:  
`@SuppressWarnings({"unchecked", "deprecation"})`
- **@FunctionalInterface** annotation, introduced in Java SE 8, indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification
- **@SafeVarargs** annotation, when applied to a method or constructor, asserts that the code does not perform potentially unsafe operations on its varargs parameter
- When this annotation type is used, unchecked warnings relating to varargs usage are suppressed.

# Annotations

## Annotations That Apply to Other Annotations:

➤ Annotations that apply to other annotations are called ***meta-annotations***. There are several meta-annotation types defined in `java.lang.annotation`.

➤ **@Retention** annotation specifies how the marked annotation is stored:

- ✓ `RetentionPolicy.SOURCE` – The marked annotation is retained only in the source level and is ignored by the compiler.
- ✓ `RetentionPolicy.CLASS` – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
- ✓ `RetentionPolicy.RUNTIME` – The marked annotation is retained by the JVM so it can be used by the runtime environment.

➤ **@Documented** annotation indicates that whenever the specified annotation is used those elements should be documented using the Javadoc tool. (By default, annotations are not included in Javadoc.)

➤ **@Target** annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to.

**A target annotation specifies one of the following element types as its value:**

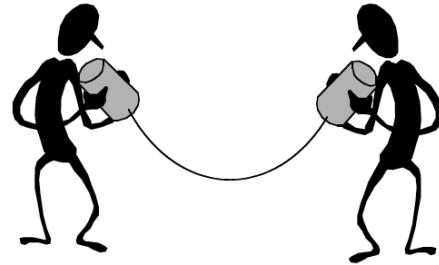
- `ElementType.ANNOTATION_TYPE` can be applied to an annotation type.
- `ElementType.CONSTRUCTOR` can be applied to a constructor.
- `ElementType.FIELD` can be applied to a field or property.
- `ElementType.LOCAL_VARIABLE` can be applied to a local variable.
- `ElementType.METHOD` can be applied to a method-level annotation.
- `ElementType.PACKAGE` can be applied to a package declaration.
- `ElementType.PARAMETER` can be applied to the parameters of a method.
- `ElementType.TYPE` can be applied to any element of a class.

➤ **@Inherited** annotation indicates that the annotation type can be inherited from the super class. When the user queries the annotation type and the class has no annotation for this type, the class' superclass is queried for the annotation type. This annotation applies only to class declarations.

➤ **@Repeatable** annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use.

# Network Programming

## Socket-based communication



Sockets are the end points of connections between two hosts and can be used to send and receive data.

There are two kinds of sockets: *server sockets* and *client sockets*.

A *server socket* waits for requests from clients.

A *client socket* can be used to send and receive data.

# Ports



A server socket listens at a specific *port*.

A port is positive integer less than or equal to 65565.

The port number is necessary to distinguish different server applications running on the same host.

Ports 1 through 1023 are reserved for administrative purposes (e.g. 21 for FTP, 23 for Telnet, 25 for e-mail, and 80 for HTTP).



# Server sockets

A server socket is an instance of the `ServerSocket` class and can be created by one of these constructors:

`ServerSocket(int port)`

`ServerSocket(int port, int backlog)`

`port`: port number at which the server will be listening for requests from clients.

`backlog`: the maximum length of the queue of clients waiting to be processed (default is 50).

Server sockets can be created only with Java applications, not applets.

# Methods of ServerSocket

`Socket accept()`

Waits for a connection request. The thread that executes the method will be blocked until a request is received, at which time the method return a client socket.

`void close()`

Stops waiting for requests from clients.

# Typical use of ServerSocket

```
try {
 ServerSocket s = new ServerSocket(port);
 while (true) {
 Socket incoming = s.accept();
 «Handle a client»
 incoming.close();
 }
 s.close();
} catch (IOException e) {
 «Handle exception»
}
```



# Client sockets

A client socket is an instance of the `Socket` class and can be obtained in two ways:

(1) On the server side as return value of the `accept()` method.

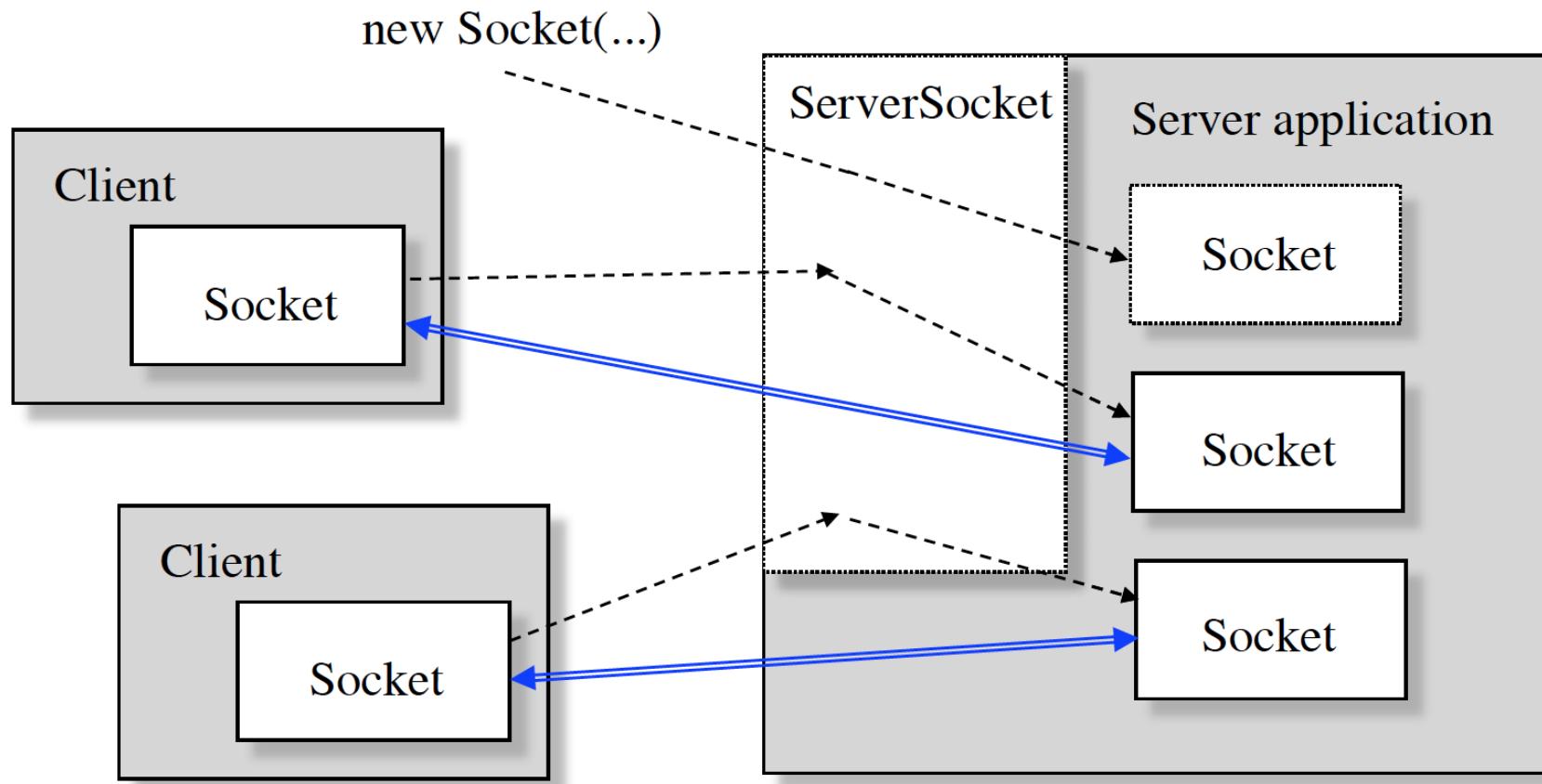
(2) On the client side by using the constructor

```
Socket(String host, int port)
```

host: the address of the host

port: the port number

# Clients' communication with a server



Communication is handled on both sides by **Socket** objects.

# Methods of Socket

`getInputStream()`

Returns an `InputStream` object for receiving data

`getOutputStream()`

Returns an `OutputStream` object for sending data

`close()`

Closes the socket connection

# Typical use of Socket

```
try {
 Socket socket = new Socket(host, port);
 BufferedReader in = new BufferedReader(
 new InputStreamReader(
 socket.getInputStream()));
 PrintWriter out = new PrintWriter(
 new OutputStreamWriter(
 socket.getOutputStream()));
 «Send and receive data»
 in.close();
 out.close();
 socket.close();
} catch (IOException e) {
 «Handle exception»
}
```

## Basic client – server program

### **MyClient.java**

```
import java.io.*;
import java.net.*;

public class MyClient {

 public static void main(String[] args) throws IOException{
 Socket sock=new Socket("localhost",6666);

 //ObjectOutputStream out=new ObjectOutputStream(sock.getOutputStream());
 //out.writeUTF("Hello Server");

 PrintWriter out=new PrintWriter(sock.getOutputStream());
 out.println("Hello Server");
 out.flush();

 out.close();
 sock.close();
 }
}
```

## **MyServer.java**

```
import java.io.*;
import java.net.*;

public class MyServer{
public static void main(String[] args) throws IOException{
 ServerSocket sos=new ServerSocket(6666);
 System.out.println("Listening on port....6666!");
 Socket sock=sos.accept();
 System.out.println("Client Connected!");

//ObjectInputStream ois=new ObjectInputStream(sock.getInputStream());
//String str=(String)ois.readUTF();

BufferedReader br=new BufferedReader(new InputStreamReader(sock.getInputStream()));
String str=(String)br.readLine();
System.out.println("Client Says="+str);

sos.close();
out.close();

}

}
```

## Client – Server, two-way communication

### **MyClient.java**

```
import java.io.*;
import java.net.*;

public class MyClient {

 public static void main(String[] args) throws IOException{
 Socket sock=new Socket("localhost",6666);
 //ObjectOutputStream out=new ObjectOutputStream(sock.getOutputStream());
 //out.writeUTF("Hello Server");

 PrintWriter out=new PrintWriter(sock.getOutputStream());
 out.println("Are you getting my message...?");
 out.flush();

 BufferedReader br=new BufferedReader(new InputStreamReader(sock.getInputStream()));

 String str=(String)br.readLine();
 System.out.println("Server Says="+str);
 out.close();
 sock.close();
 }
}
```

## **MyServer.java**

```
import java.io.*;
import java.net.*;

public class MyServer{
public static void main(String[] args) throws IOException{
 ServerSocket sos=new ServerSocket(6666);
 System.out.println("Listening on port....6666!");
 Socket sock=sos.accept();
 System.out.println("Client Connected!");

 //ObjectInputStream ois=new ObjectInputStream(sock.getInputStream());
 //String str=(String)ois.readUTF();

 BufferedReader br=new BufferedReader(new InputStreamReader(sock.getInputStream()));
 String str=(String)br.readLine();
 System.out.println("Client Says="+str);

 PrintWriter out=new PrintWriter(sock.getOutputStream());
 out.println("Yes...I am !");
 out.flush();

 sos.close();
 out.close();

}
```

T<sub>1</sub>  
Thank You!!!

**Sadhu Sreenivas**