INTRODUCTION AND REVIEW

Course: Object Oriented Programming (CS F213)

PROF. ANITA AGRAWAL

COURSE INFORMATION

Course ID: CS F213- Object Oriented Programming

Instructor In-charge- Prof. Anita Agrawal

- Office- A404
- Email- aagrawal@goa.bits-pilani.ac.in

Refer to General Handout appended to Time-Table

PREREQUISITES

Successful completion of “ CS F111: Computer Programming”

LEARNING OUTCOMES

Upon successful completion of this course, students will have a thorough understanding of object oriented analysis and design process, and will be able to demonstrate object-oriented concepts in Java programming language.

BOOKS AND REFERENCE MATERIAL

Textbooks-

- T1: The Complete Reference Java J2SE, 5th Edition, Herbert Schildt, Tata McGraw Hill and Publishing 2005
- T2: Objects First with Java: A Practical Introduction Using BlueJ, David J. Barnes and Michael Kolling, Pearson Education, 5th Edition, 2012.

Reference Material

- R1: Head First Java, Bert Bates, O'Reilly, 2nd Edition, 2005.
- R2: Core Java Volume I - Fundamentals, Cay Horstmann, Pearson Education, 8th Edition 2008

COURSE PLAN AND MODULES

- ☐ Introduction and review
- ☐ Module 1: An Overview of OOP
  - ☐ Class definition and Object
  - ☐ Principles of OOP
  - ☐ Introduction to Java program syntax
  - ☐ Compiling and execution of Java program
- ☐ Module 2: An Overview of Java
  - ☐ Primitive data types
  - ☐ Type conversion and casting
  - ☐ Arrays, Operators, and Control statements
  - ☐ Minor differences between C and Java
- ☐ Module 3: Introducing Classes
  - ☐ Classes and Objects
  - ☐ Constructors and Methods
  - ☐ Garbage collection
- ☐ Module 4: UML, Sequence, and State diagrams
- ☐ Module 5: Deeper into Classes
  - ☐ Method overloading
  - ☐ Argument passing
  - ☐ Access specifiers
  - ☐ Static and final
  - ☐ Nested and inner classes

- ❑ Module 6: String Class and Variable Length Arguments

- ☐ Module 7: Inheritance
  - ☐ Super and abstract classes
  - ☐ Instance variable hiding
  - ☐ Multilevel hierarchy
  - ☐ Method overriding

- ☐ Module 8: Packages
  - ☐ Importing and Creating packages
  - ☐ Access protection
  - ☐ Interfaces

- ❑ Module 9: Exception Handling
  - ☐ Exception types
  - ☐ Try and catch, Nested try statements
  - ☐ Java's built-in exceptions
  - ☐ Keywords: throw, throws, and finally

- ☐ Module 10: Threads
  - ☐ Creating new threads
  - ☐ Thread synchronization

- ☐ Module 11: File and I/O
  - ☐ File handling in Java
  - ☐ I/O Classes and Interfaces
  - ☐ Stream classes

- ☐ Module 12: Advanced Topics in OOP and Java

EVALUATION SCHEME

Component	Type	Weightage Percentage (marks)out of 200	Duration	Date, day & Time
Mid-sem Exam	Closed Book	30% (60)	1.5 hours	16/03/2019, Saturday, 2 to 3:30 pm
Regular Labs	Open Book	30 % (60)	2 hours	Thursday, 10-11 hr.
Comprehensive Exam	Closed Book	40% (80)	3 hours	14/05/19, Tuesday (FN),

Best 8 out of 10 evaluated labs shall be considered for grading.

CHAMBER CONSULTATION HOUR

Monday 3rd hr.

MAKE-UP POLICY

1. Make-ups shall be allowed only in genuine cases, on a case-by-case basis and on proper justification. Prior permission from the IC is necessary.
2. No make-up for the lab sessions.
3. Missing of any one of the three components may be graded as **NC**, despite of satisfying minimum necessary criteria in other components.

MALPRACTISE

Any attempt of cheating or plagiarism in tests or labs will attract disciplinary committee action.

OTHER USEFUL DETAILS

- All notices concerning this course will be displayed on moodle course page.

- Important notices/announcements will be mailed to you directly in addition to posting on moodle.

Object-Oriented Programming- An Introduction

Lecture 2
9/1/19

Prof. Anita Agrawal

OOP why????

- Computer scientists have struggled for decades to design new languages and techniques for writing software.
- Unfortunately, experience has shown that writing for large systems was virtually impossible.
- Small programs seemed to be no problem, but scaling to large systems with large programming teams could result in $100M projects that never worked and were thrown out.

- The only solution seemed to lie in writing small software units that communicate via well-defined interfaces and protocols like computer chips.

- The units must be small enough that one developer can understand them entirely and most importantly the units must be protected from interference by other units so that programmers can code the units in isolation

- The Object-Oriented paradigm fits these guidelines as designers represent complete concepts or real world entities as objects with approved interfaces for use by other objects.

Object Oriented Programming

- Four major principles:
- 1. Data Abstraction
- 2. Encapsulation
- 3. Polymorphism (dynamic binding)
- 4. Inheritance

- For many tasks, object-oriented programming has proven to be a very successful paradigm.
- Interestingly, the first object-oriented language (called Simula), was designed in the 1960's, but object-oriented programming has only come into fashion in the 1990's.

Other HL Programming languages/Procedural Programming

- Good for simple programs- Structured and splits tasks into smaller ones.
- But bigger the program, more lengthy and tougher to manage by programmers
- Example: C programming language

OOP ???

- **O**bject-**O**riented **P**rogramming is a programming paradigm based on the concept of "objects" and "classes"
- Entire world depends on **classes** and **objects** which are at the heart of the concept

Object???

- **Object**: is a software bundle of related **state** and **behavior**.
- Software objects are often used to model the real-world objects that you find in everyday life.

Class???

- A class is a blueprint or prototype from which objects are created.
- A class models the state and behavior of the object.

Object Oriented Programming Languages

A methodology of programming used by popular programming languages like Java, C++, C#, Ruby, etc.

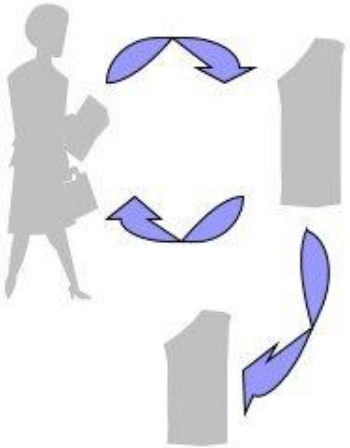
Procedural Programming vs Object Oriented Programming

- The unit in procedural programming is ***function***, and unit in object-oriented programming is ***class.***

- **Procedural** programming concentrates on creating **functions**, while **object-oriented programming** starts from isolating the **classes**, and then look for the methods inside them.

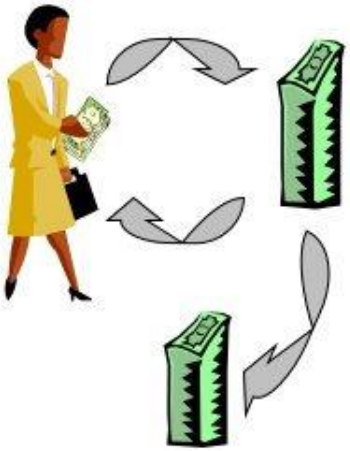
Procedural vs. Object-Oriented

- Procedural



Withdraw, deposit, transfer

- Object Oriented



Customer, money, account

Object.....

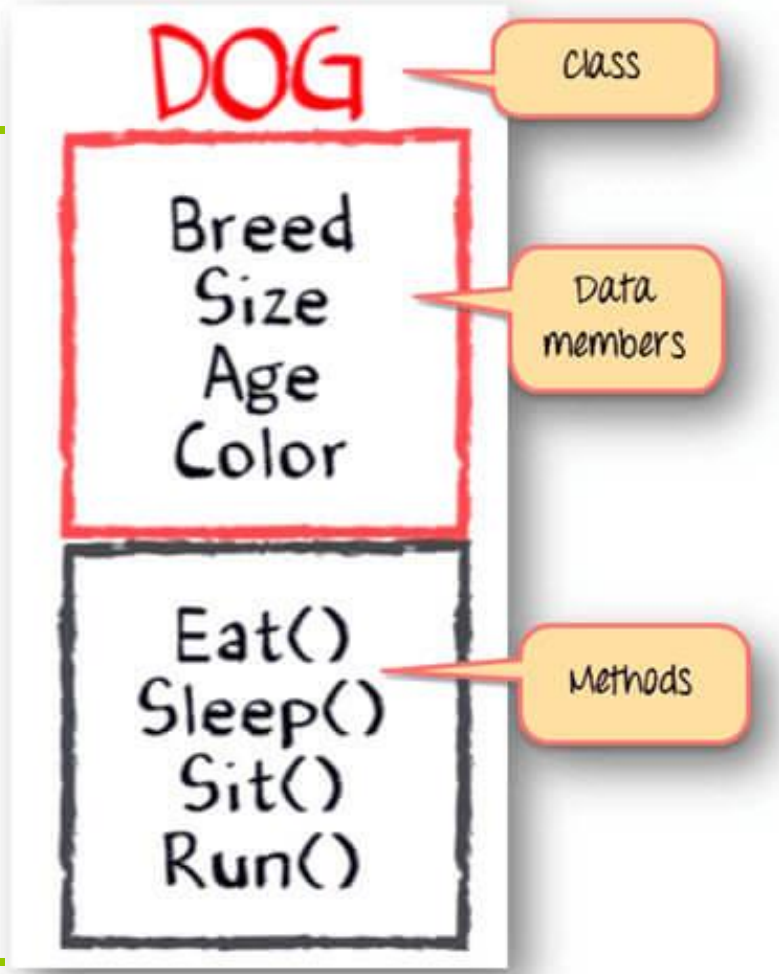
- It is a basic unit of Object Oriented Programming and represents the real life entities.
- It consists of :
  - State
  - Behavior
  - Identity

- **State-** It is represented by **attributes** of an object. It also reflects the properties of an object.
- **Behavior** - It is represented by **methods** of an object. It also reflects the response of an object with other objects.
- **Identity**-It gives a unique name to an object and enables one object to interact with other objects.

- **Attributes:**
  - These are the **instance variables**. Each object will have its unique set of instance variables.
  - An object's state is created by the values assigned to these instance variables

- **Method:**
  - It is basically a behavior.
  - A class may contain many methods.
  - It is in methods that the logics are written, data is manipulated and all the actions are taken

Class members



An example of an Object- Dog

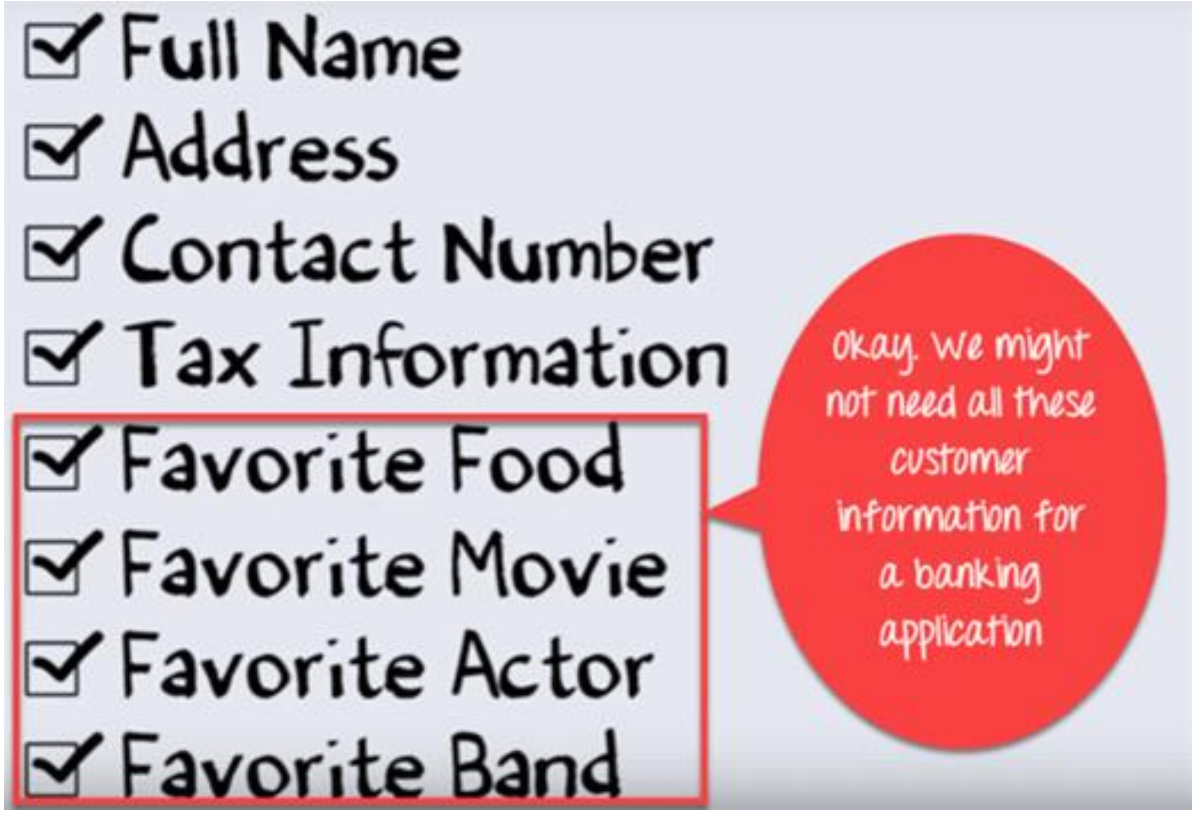
Identity	State/Attributes	Behaviors
Name of dog	*Breed* *Age* *Color*	*Bark* *Sleep* *Eat*

The 4 fundamental OOPS Concepts

1. Data Abstraction
2. Encapsulation
3. Polymorphism (dynamic binding)
4. Inheritance (particular case of polymorphism)

Data Abstraction

- It deals with **hiding the details** and **showing the essential things** to the user.
- In Java, abstraction is accomplished using **Abstract classes** and interfaces.
- An example of abstraction- A Banking application, where only required data fields are revealed to the user


Full Name
Address
Contact Number
Tax Information
Favorite Food
Favorite Movie
Favorite Actor
Favorite Band
Okay. We might not need all these customer information for a banking application

Encapsulation

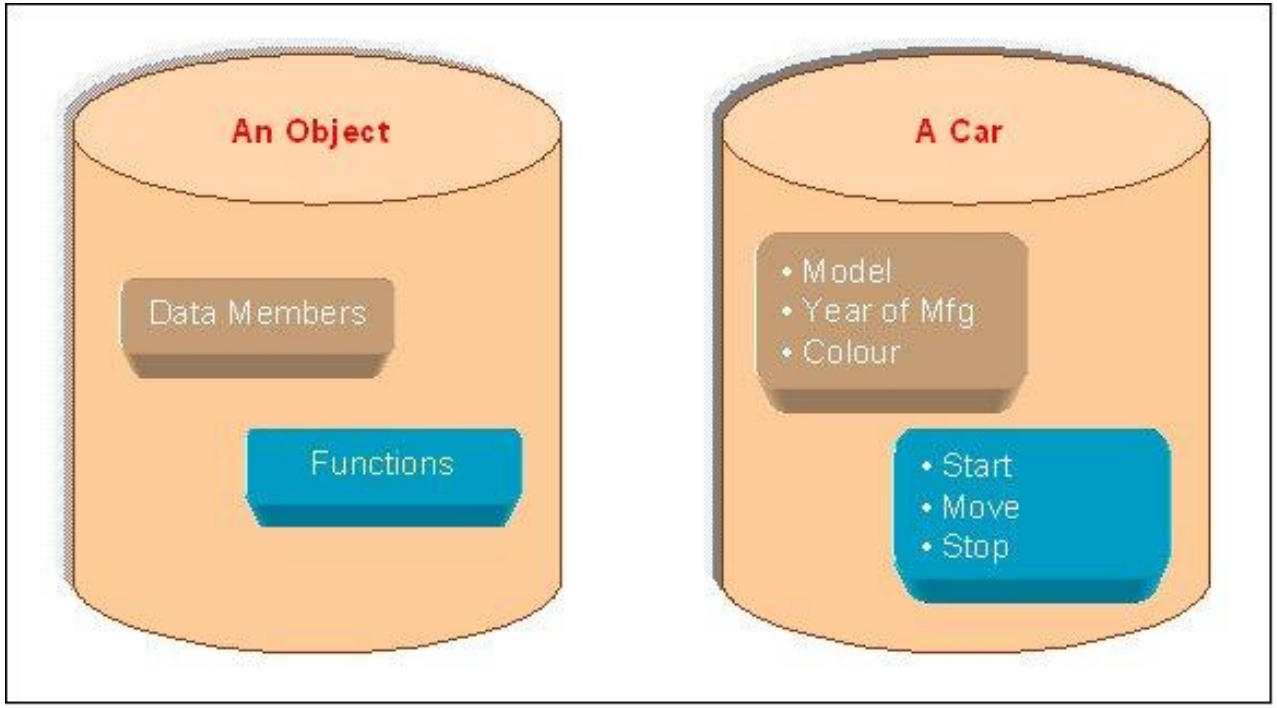
Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.

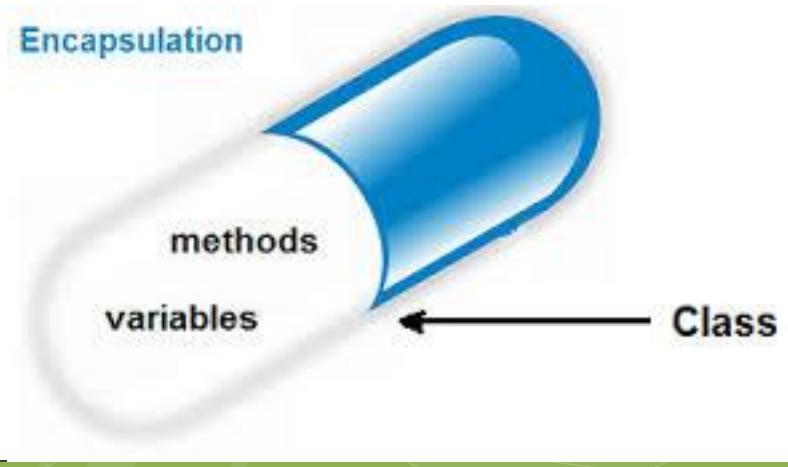
The variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class

Encapsulation and classes

- Class contains: Data → Member/Instance variables
  Code → Member methods

- Class encapsulates complexity


An Object
Data Members
Functions
A Car
• Model
• Year of Mfg
• Colour
• Start
• Move
• Stop


Encapsulation
methods
variables
Class

Inheritance

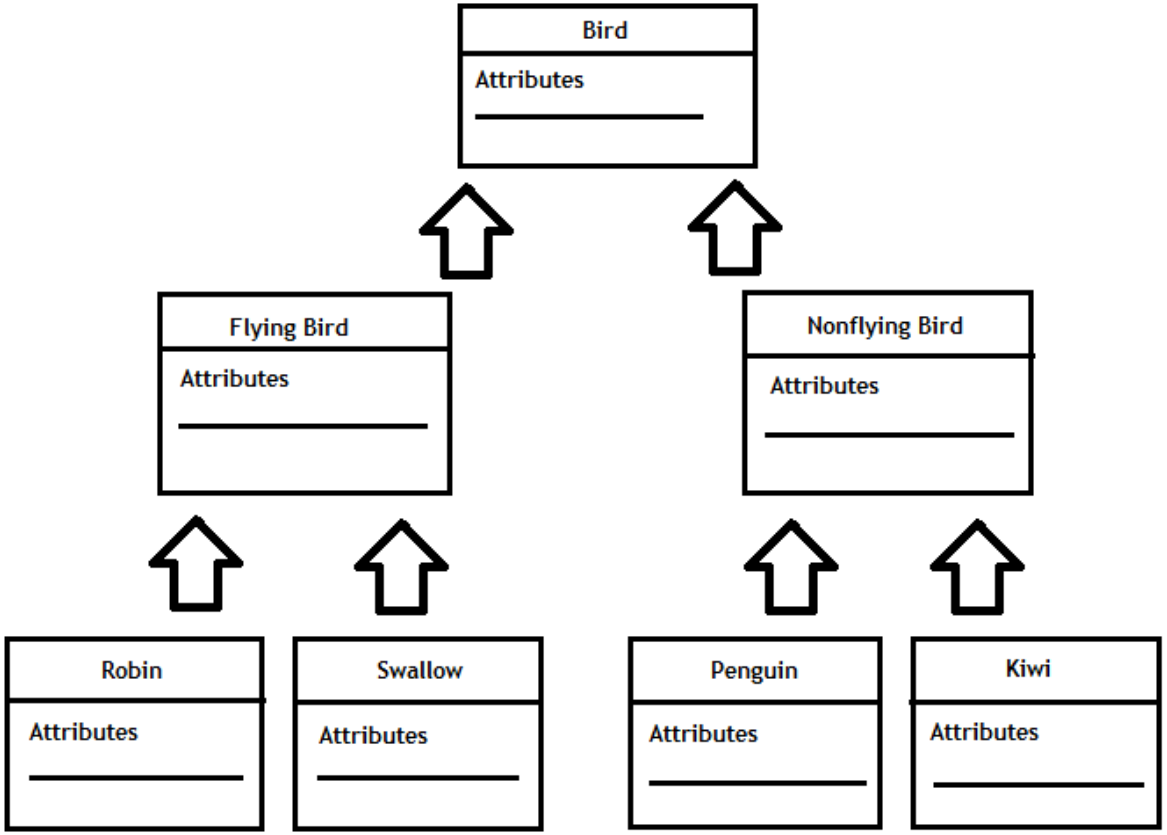
- A mechanism in which one class acquires the property of another class.
  - **Example, a child inherits the traits of his/her parents.**

With inheritance, the fields and methods of the existing class can be reused.

- An inherited class is called a **subclass** of its parent class or super class.

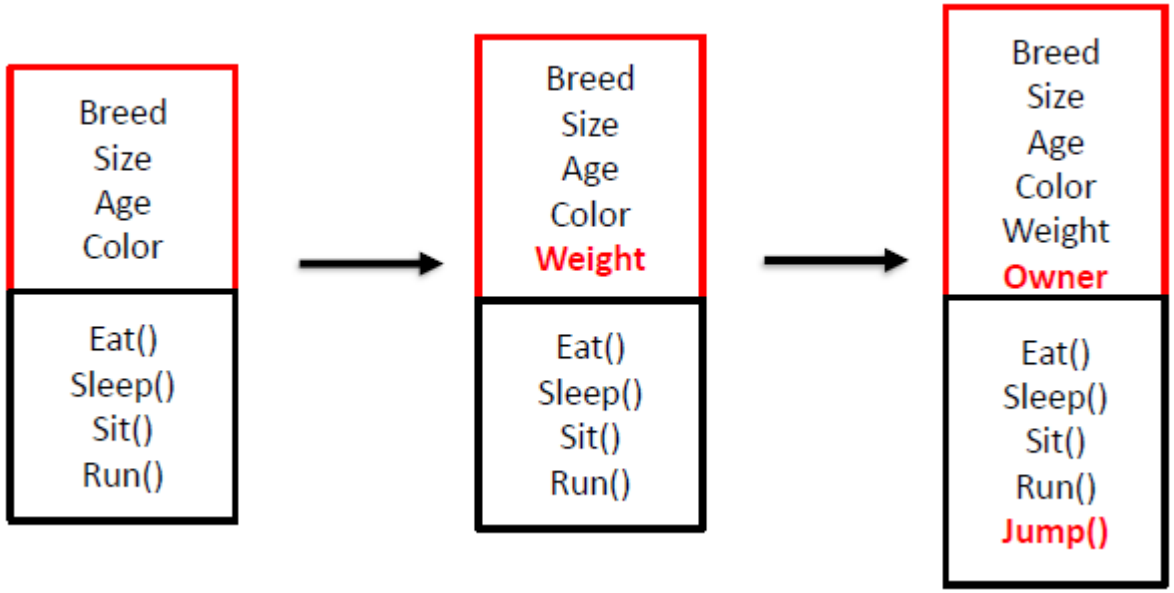
- It can be of different types- Single, Multiple, Multilevel, Hierarchical and Hybrid

Real World example of inheritance



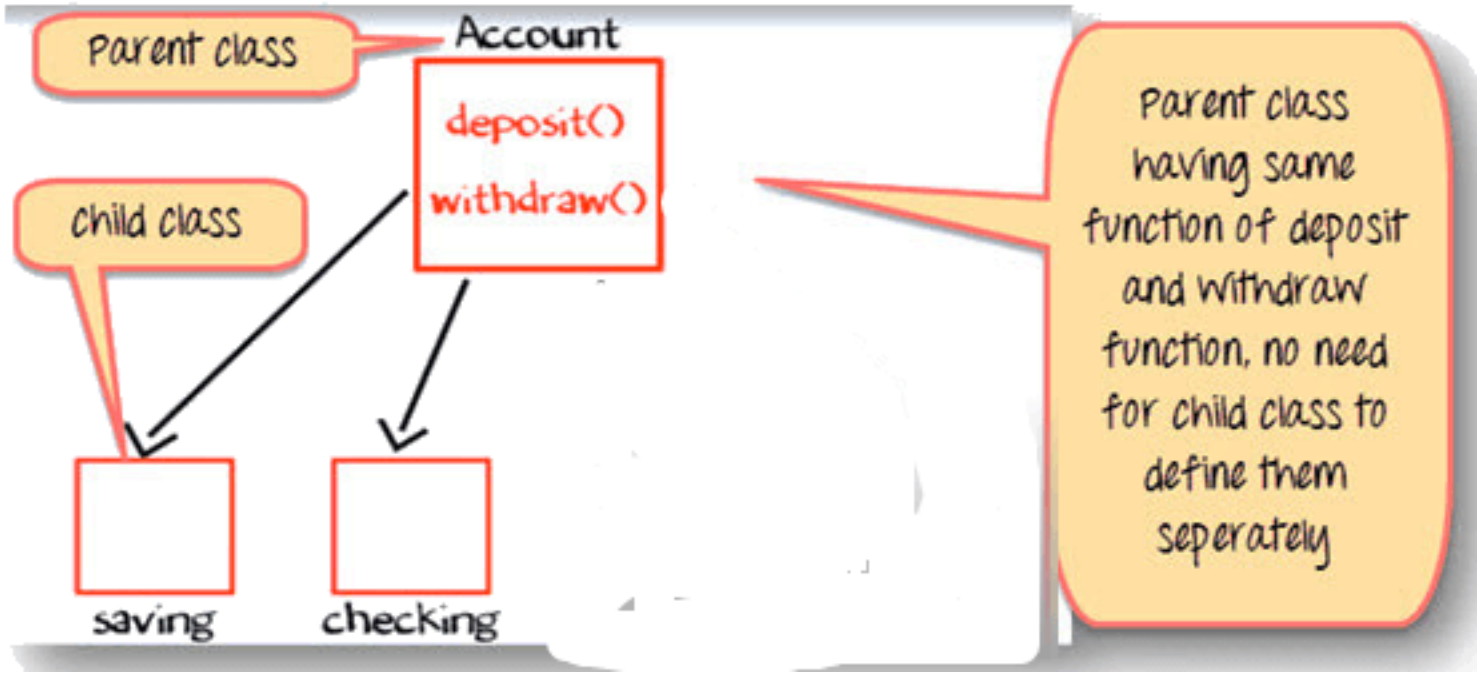
Another example of Inheritance:

One class acquires properties of another class



Supports hierarchical classification

Real World Example in a banking system



Polymorphism

- It is a OOP concept where one name can have many forms, i.e. variable, function or object can take on multiple forms.
- In a real world example, a person at a same time can have different characteristic.
- Example- Calculate the Area-

Lecture 3 A
Data Types and Variables

Course: Object Oriented Programming (CS F213)

Prof. Anita Agrawal
BITS Pilani-K.K.Birla Goa campus

Quick recap..

- Software objects are conceptually similar to real-world objects: they too consist of **state** and **related behavior.**

- An object stores its **state** in ***fields*** (variables in some programming languages) and exposes its **behavior** through ***methods*** (functions in some programming languages).

- **Methods** operate on an **object's internal state** and serve as the primary mechanism for object-to-object communication.

Today's session....

- Objects store their state in **fields**. However, the Java programming language uses the term "**variable**" as well.

- In this Lecture……..
  - **Variable naming rules and conventions**
  - **Basic data types**
  - **Default values** and
  - **Literals.**

Types of variables

- 4 types:
  - Instance variables
  - Class variables
  - Local variables
  - Parameters

Types of variables

- **Instance variables (non-static fields):**
  - Objects store their individual states in non-static fields
  - Their values are unique to each instance of a class (to each object, in other words)
- **Example:** the currentSpeed of one bicycle is independent from the currentSpeed of another

Types of variables

- **Class variables (static fields):**
  - This tells the compiler that there is exactly one copy of this variable in existence, regardless of how many times the class has been instantiated.

- **Example:** A field defining the number of legs for a dog could be marked as static since conceptually the same number of legs will apply to all instances.

```
static int legs = 4;
```

Types of Variables

- **Local Variables:**
  - Used by methods.
  - A method will often store its temporary state in local variables.
  - Only visible to the methods in which they are declared; they are not accessible from the rest of the class.
- Example:

```
void calculate()
{
        int count = 0;
….}
```

Local variable

Types of Variables

- **Parameter Variables:**
  - Parameters are always classified as **"variables"** not "fields".
  - This applies to other parameter-accepting constructs as well (such as constructors and exception handlers)
  - Example: The signature for the main method is public static void main(String args []).
    - Here, the **args** variable is the **parameter** to this method.

Primitive Data type

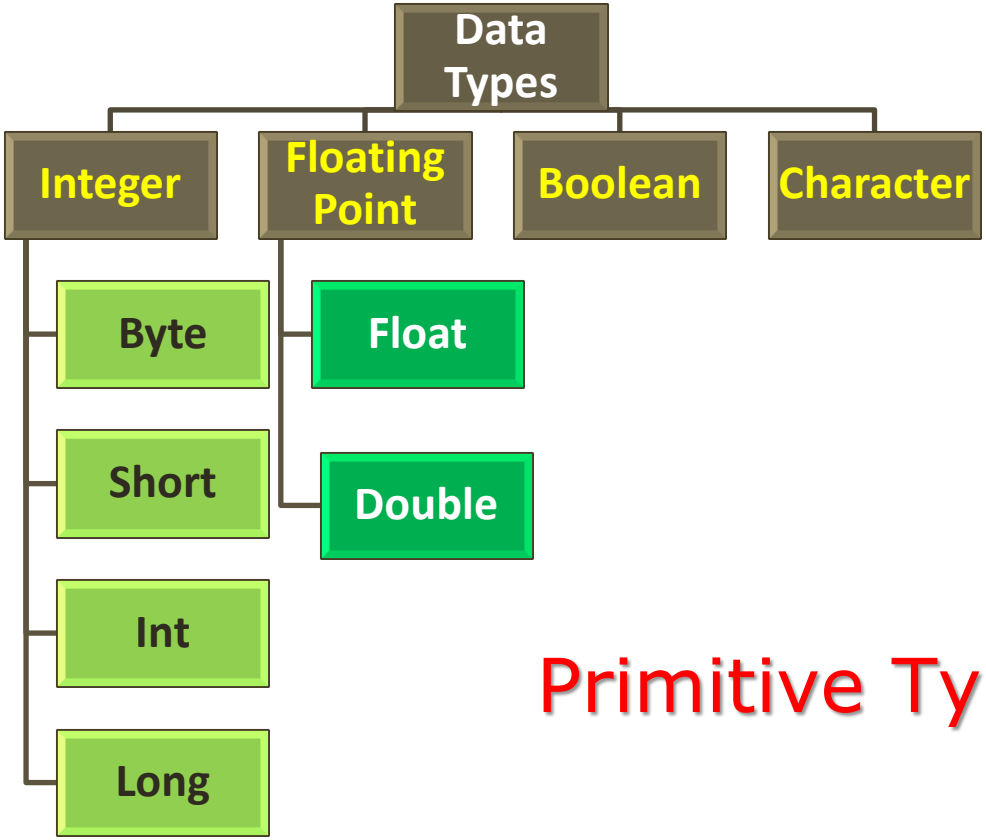
- The Java programming language is **statically-typed**, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name.

- Ex: **int** legs = 2;

- A variable's data type determines the **values** it may contain, plus the **operations** that may be performed on it.

- Primitive types are special data types built into the language; they are not objects created from a class.
- A **primitive type** is predefined by the language and is named by a **reserved keyword**.
- There are **eight** primitives in Java.
- All types have **strictly defined range**
- This make Java program **“Portable”**

- –In C/C++, **size** of an integer depends on the **particular architecture**



Primitive Types

Primitive Data Types in Java: Integers

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

- All of the above are stored as signed two's complement numbers
- Java does not support unsigned positive only integers

Primitive data types in Java: floating-point numbers

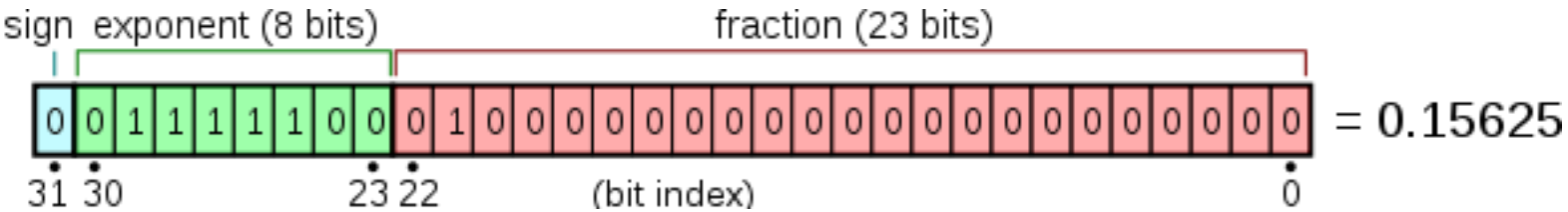
Also known as real numbers • Two kinds of floating point types to store: –

- Float type (Single precision)
- Double Type (Double precision)

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

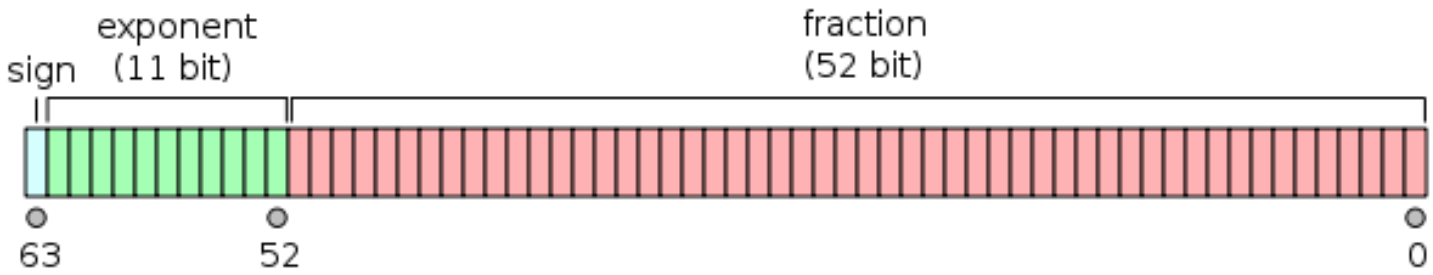
Primitive Data Types in Java: floating point numbers

- **IEEE 754 single-precision binary floating-point format:**
- Sign bit: 1 bit
- Exponent width: 8 bits
- Significand precision: 23 bits (explicitly stored)



Primitive Data Types in Java: floating point numbers

- **IEEE 754 double-precision binary floating-point format:**
- Sign bit: 1 bit
- Exponent width: 11 bits
- Significand precision: 52 bits (explicitly stored)



Java – C: A comparison

- Java is a **strongly** typed language.
- C is a **weakly** typed language

Example: (In C…)

```
int main()
 {
        int i= 3.0;
        printf("Value of i is %d", i);
        return 0;
}
```

Output: Value of i is 3

- Example: (In Java)

```
class Example
{
        public static void main(String args[])
            {
               int i= 3.0;
                System.out.println("value of i: "+i);
            }
 }
```

Type mismatch throws an error. (fldemo)

error: incompatible types:

Possible lossy conversion from double to int

int i=3.0;

Char type

- Java uses Unicode to represent characters.
- Unicode defines fully international character set: English, Latin, Greek, and many more
- The minimum value of char data type is '\u0000' (0). The maximum value of char data type is '\uffff'.
- It supports ASCII values also 0-255

Boolean Type

- Takes two values- true and false
- Returned by relational operators and used by conditional expressions

String type

- In addition to the 8 primitive data types, the Java programming language provides special support for character strings via the java.lang.String class.
- It is technically not a primitive data type.
- String objects are immutable.

Literals

Literals

- A ***literal*** is the source code representation of a fixed value
- They are represented directly in your code without requiring computation.
- it's possible to assign a literal to a variable of a primitive type.

Literals-Integer representation

- An integer literal is of type **long** if it ends with the letter L or l; otherwise it is of type **int**

- Integer literals can be expressed by the following number systems:

  - Decimal
  - Octal
  - Hexadecimal
  - Binary

- For general purpose-programming, we use the decimal number system

Syntax to be followed for other number systems

- int decval: 1238 //number in decimal
- int Octval: 01237 //number in octal
- int Hexval: 0x12ABC //number in hexadec
- int binval: 0b10110 //number in binary

Literals: Floating point

- Floating-point literals are written with a decimal point.
- Can be represented as
  - –Standard Notation: 3.1234, 56.778
  - –Scientific Notation: 6.022E23, 1234E-13, 23e+100

- Floating-point literals are by default **double**
  - For example, 5.0 is considered a double value, not a float value.

- To store a literal as **float,** we have to append *F* or *f* to the constant

- **float f = 3.145; // Error**
- **float f = 3.145f;// Correct way**

Boolean literals

- Used to represent logical values: **true** and **false**
- **True** and **False** do not convert into numerical representation
- True≠1 and False≠0 (Unlike C/C++)
- True and False can be only assigned to boolean variable

Lecture 2
Introduction to Java

Course: Object Oriented Programming (CS F213)

PROF. ANITA AGRAWAL
BITS PILANI, K.K. BIRLA GOA CAMPUS

In today's session...

INTRODUCTION TO JAVA

Java Program

- A Java program is defined as a collection of objects that communicate via invoking each other's methods.

Anatomy of A Java Program

A. Reserved words / Keywords
B. Classes
C. Package
D. Modifiers
E. Statements
F. Blocks
G. Methods
H. The main method
I. Comments

My first program


```
class myfirstjavaprogram
{
    public static void main(String args[])
    {
        System.out.println("this is my first java program");
    }
}
```

Breaking down the program

- "class" keyword is used to declare a new class.
- "Myfirstjavaprogram" is the identifier which is the name of the class.
- The program begins executing at public static void main
- public:
  - It is an example of an access specifier.
  - Allows programmer to control the visibility of class members
  - Members may be accessed from outside the class in which they are declared.

- main() be declared as public since it will be called by outside it's class

Breaking down the program (Contd.)

- **static:-**
  - Allows main() to be called without creating an object of the class.
  - It is necessary since main() is called by JVM before any objects are made
- **void:-**
  - Defines the return type, in this case it is void

- **main:-**
  - The system locates and runs the main method for a class when you run a program.
  - Other methods get executed when called by the main method.

- **String args[ ]:**

  –Declares a variable (object) name **args**

  –It is an array of String class instances

  –args[ ] receives any command line arguments during program execution

System.out.println

- It is a *method*: a collection of statements that performs a sequence of operations to display a message on the console.

- It is used by invoking a statement with a string argument.

- The string argument is enclosed within parentheses. In this case, the argument is "This is my first java program!"

- You can call the same println method with a different argument to print a different message.

More specifically....

System.out.println ("This is my first Java program");

- This line displays the string on screen followed by a new line
- println () is a built-in method
- System is a pre-defined class that provides access to the system
- out is the output stream

Blocks

- A pair of braces in a program forms a block that groups components of a program.

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Welcome to Java!");
  }
}
```

Class block

Method block

Comments


```
//my first java program
class myfirstjavaprogram
{
    public static void main(String args[])
      {

            System.out.println("this is my first java program" );

       }
}
```

```
// This is a single line comment
/* These are multiple
        line comments */
```

Lexical constraints

- Whitespace – Java is a free-form language (indentation is not necessary but desirable).
  - As long as there is one whitespace character between each token – Space, tab, and newline.
- Identifiers – Used for class, method, and variable name.
  - Combination of upper and lowercase letters, and numbers – Underscore_ and $ sign are allowed
- Literals – A constant value in Java is called literal
- Comments – /* */ and //

List of Separators

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

Reserved words/ Keywords in Java

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

Important points

- The name of the source file is very important.
- In previous example, the name of the file will be **myfirstjavaprogram.java**
- The name of the class should match the name of file that holds the program.
- A source file is officially called **compilation unit.**

- It is a text file that contains one or more class definitions.
- For Java compiler, source file must have **.java** extension.
- In Java, all code must reside inside a class
- Java is a **case-sensitive** programming language.

Example 2: If control statement


```
//if control statement

//if statement
class if_eg
{
        public static void main(String args[])
        {
                int n=10;
                if(n%2== 0)
                        System.out.println("The number is even");
                else
                        System.out.println("The number is odd");
        }
}
```

History of Java

- Initially called ‘Oak’, but after a while (1995) came to be known as ‘Java’, over coffee.

- **J**ames Gosling, **A**rthur **V**an Hoff and **A**ndy Bechtolsheim

Salient Features of Java

- Object-oriented programming language
- Structured in terms of ***classes***, which group data with operations on that data
- Can construct new classes by ***extending*** existing ones

Salient features of Java

- **Platform Independent:**

Unlike many other programming languages including C and C++, when compiled, it is not compiled into a code which is platform specific, rather into "**platform-independent byte code".**

- It runs on a variety of platforms such as Windows, Mac OS, and the various versions of unix

Salient Features of Java

- **Simple:**

Easy to learn

- **Secure:**

Enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption

Salient Features of Java

- **Architecture-neutral:**

  Generates an architecture-neutral object file format which makes the compiled code executable on many processors with the presence of Java runtime system.

- **Portable:**

  Being architecture neutral and having no implementation dependent aspects of the specification makes it portable

Salient Features of Java

- **Robust:**

  Eliminates error-prone situations by emphasizing mainly on compile time error checking and runtime checking

- **Multithreaded:**

  Programs can be written that can perform many tasks simultaneously

- **Interpreted:**

  Java code is translated on the fly to native machine instructions and not stored anywhere.

Salient Features of Java

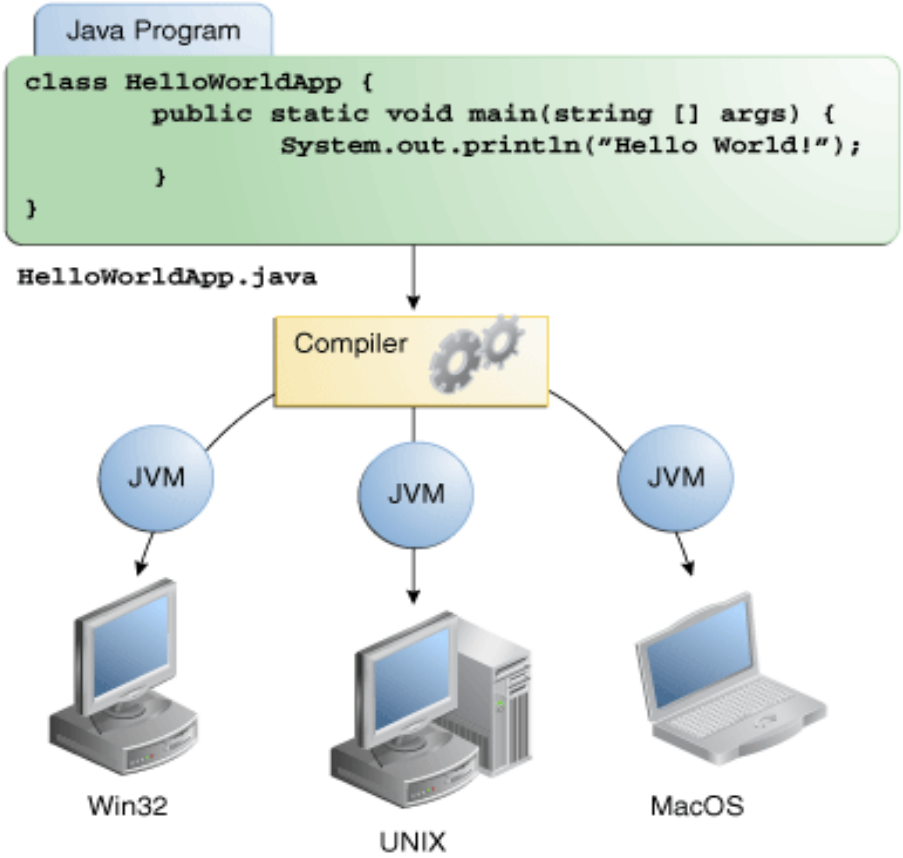
- **High Performance:**

  With the use of Just-In-Time compilers, Java enables high performance

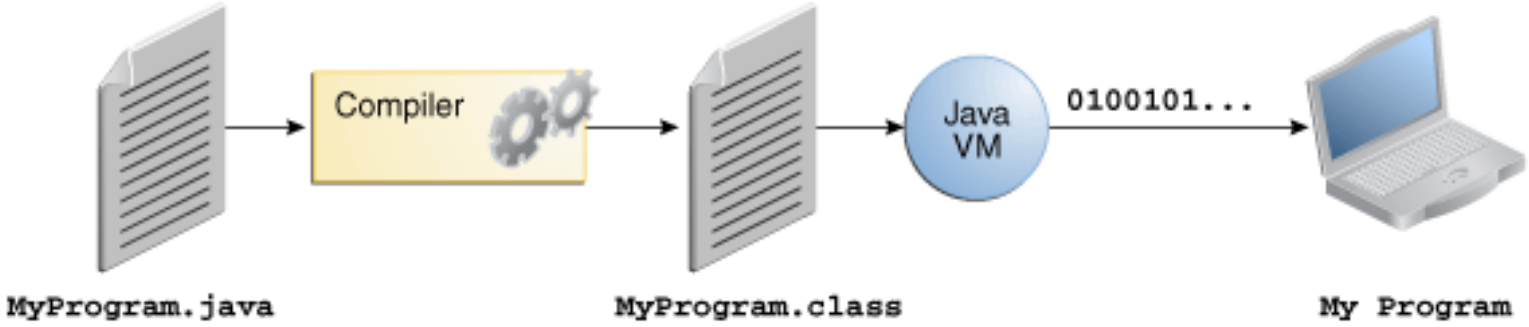
- **Dynamic:** More dynamic than C or C++
  - **Can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.**

ByteCode

- Output of Java Compiler is not executable code but is a **Bytecode**.
- Bytecode is a highly optimized set of instructions to be executed by Java Runtime System (Java Virtual Machine- JVM).
- JVM is the interpreter for Bytecode -> Convert into machine code for execution.


Java Program
class HelloWorldApp {
public static void main(string [] args) {
System.out.println("Hello World!");
}
}
HelloWorldApp.java
Compiler
JVM
JVM
JVM
Win32
UNIX
MacOS

Java program: Stages of execution



Bytecode- Contd.

- Advantages:
  - It makes it easier to execute a program
  - Only the JVM needs to be implemented
- Disadvantages:
  - Slower performance

Requirements

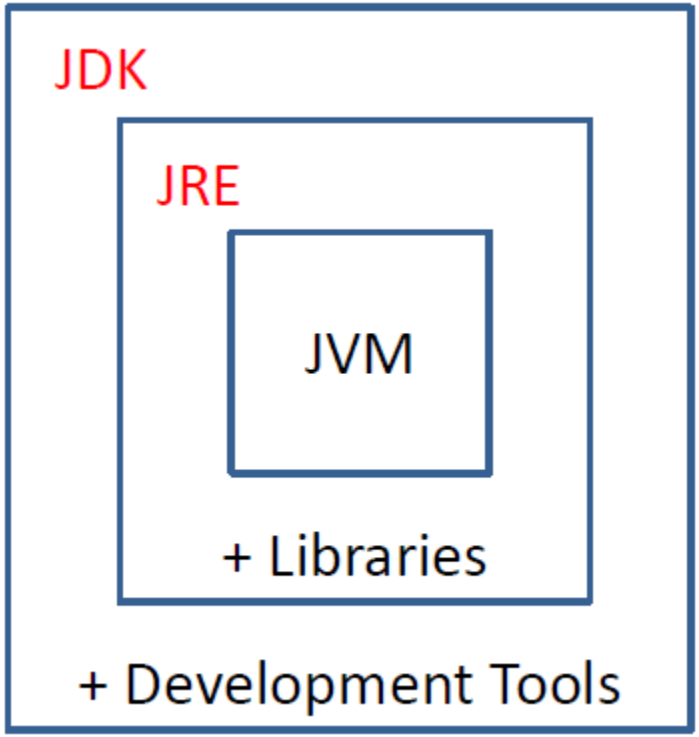
- A text editor
- The Java Development Kit (JDK)
  - Includes the Java Compiler
- The Java Runtime Environment (JRE)
  - Includes Java Virtual Machine (JVM)

JVM vs. JRE vs. JDK

- JVM – Java Virtual machine(JVM) – JVM is responsible for executing the java bytecode line by line hence it is also known as interpreter – JVM allows Java to be a "portable language" (write once, run anywhere)

- JRE – Java Runtime Environment – An installation package which provides environment to only run (not develop) the java program onto your machine. JRE includes JVM and libraries – JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.

- JDK – Java Development Kit – The JDK is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing programs and applications.


JDK
JRE
JVM
+ Libraries
+ Development Tools

Compiling a Java Program

- Execute the java compiler (**javac**)
  - –C:\>javac myfirstjavaprogram.java
- Javac compiler will create myfirstjavaprogram.**class** file
- .class file contains the **bytecode** version of the program
- Java bytecode is not an executable code
- To execute a bytecode, we need JVM

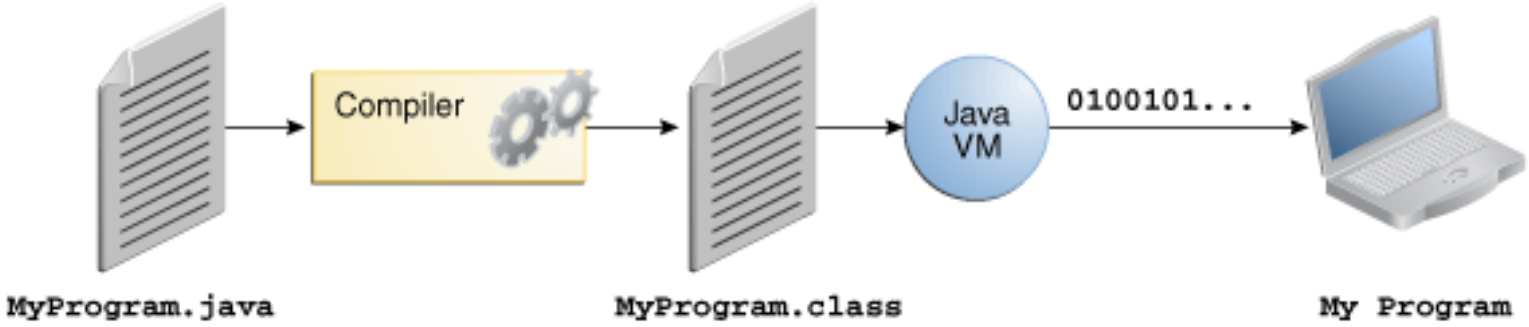
Executing a Java Program

- To run the program, we need Java Application Launcher called **java**
  - –C:\>java myfirstjavaprogram (not myfirst… .java)

- When the program is run, the following output will be displayed:

  This is my first Java program

Java program: Stages of execution



JDK editions

A. Java Standard Edition (J2SE)

   J2SE can be used to develop client-side standalone applications or applets.

B. Java Enterprise Edition (J2EE)

   J2EE can be used to develop server-side applications such as Java servlets and Java ServerPages.

C. Java Micro Edition (J2ME).

   J2ME can be used to develop applications for mobile devices such as cell phones.

Next: ……Data Types and Variables

Lecture 3B
Data Types and Variables

Course: Object Oriented Programming (CS F213)

Prof. Anita Agrawal
BITS Pilani-K.K.Birla Goa campus

Variables

- A variable is a named memory location capable of storing data

- We can also store data in simple variables, which represent data only, without any associated methods

Literals

Character Literals

- Represented by enclosing in single quotes
  - Example : char c = ‘A’;

- 16 bit values can be converted into integers and manipulated with integer operators, by enclosing in ‘ ‘.
  - Example : char c = ‘A’;

    c++; // c now contains ‘B’
  - Example:

    char letter = '\u0051';

  - Visible ASCII characters: ‘A’ ‘k’ ‘6’ ‘@’

Escape Sequences in Java

CHARACTER ESCAPE SEQUENCE	DESCRIPTION
'\n'	A linefeed
'\r'	A carriage return
'\f'	A form feed
'\b'	A backspace
'\t'	A tab
'\\'	A backslash
'\"'	A double quote
'\''	A single quote

There are only 8 escape sequences in Java. You cannot define your own character escape sequences

String type

- In addition to the 8 primitive data types, the Java programming language provides special support for character strings via the java.lang.String class.
- It is technically not a primitive data type.
- String objects are immutable.

String Literals

- Strings are implemented as objects rather than an array of characters

- Sequence of characters enclosed in a pair of double quotes.
  - Example: “Hello World!”

  - Example: “These are \n two lines”

  - Example: “\”This is shown in Quotes\”””

- Write down a Java program to print the following:

g

Java8

Example char

- class stExample

```
{
public static void main(String[] args)
    {
        char myChar = 'g';
        char newLine = '\n';
        String myString = "Java 8";
        System.out.println(myChar);
        System.out.println(newLine);
        System.out.println(myString);
    }
}
```

Example char

- class stExample

```
{
public static void main(String[] args)
    {
        System.out.println('g');
        System.out.println('\n');
        System.out.println("Java8");
    }
}
```

Example double: What will be the output of the following code??

```
class DoubleExample
 {
   public static void main(String args [])
        {
            double d =41.2;
            float f = 41.2F;
            double ds = 1.836e3;

        System.out.println(d);
        System.out.println(f);
        System.out.println(ds);
         }
}
```

Output

41.2

41.2

1836.0

Default values

- It's not always necessary to assign a value when a field is declared.
- Fields that are declared but not initialized will be set to a reasonable default by the compiler depending on the data type.
- Relying on such default values, however, is generally considered bad programming style.

Default values of the data types

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

Example: What will be the output of this code?

```
class trial
{
        public static void main(String[] args)
    {
         char c= 'A';
        c= 'B';
         System.out.println(c);
     }
}
```

Output: B

Example: What will be the output of this code?

```
class trial
{
        public static void main(String[] args)
    {
        char c= 'A';
         int i;
        c= 'B';
        System.out.println(c);
          System.out.println(i);

     }
}
```

Output: B????

Example: What will be the output of this code?

```
class trial
{
        public static void main(String[] args)
    {
         String str= “Hi”;
        str= “hello”;
         System.out.println(str);
     }
}
```

Output: hello

Type Conversion

- It is possible to assign the value of one type to the variable of other type, both implicitly (automatically) or explicitly.

- If two types are compatible, automatic conversion is performed.
  - For example, int to long int.

- For incompatible types like char-int or double-byte, this has to be done using type casting.

Automatic Type Conversion

- **Automatic conversion happens when:**
  - **Two types are compatible**
  - **Destination type is larger than the source type**

Automatic type conversion conditions:

- Two types are compatible
  - **Numeric types (integer and floating-point) are compatible with each other**
  - **Numeric types are not automatically converted to char or Boolean**
  - **char and boolean are not compatible with each other**

- Destination type is larger than the source-
- **Widening conversion happens**

Type casting

- This is done when the Destination type is smaller than the source type, i.e. narrowing conversion occurs.

- For example,

```
int i;
byte b;
b = (byte) i; //type casting
i = b;        //type casting not required
```

Example revisited

```
class  autoconvert {
public static void main(String arg[])
{
int i=5;
byte b=10;
short s=15;
long l=20;
i=i+1;
b=b+1;
s=s+1;
i=i+1;
System.out.println("i: "+i+" b: "+b+" s: "+s+" i: "+l);
}
```

Example revisited

```
class  autoconvert {
public static void main(String arg[])
{
int i=5;
byte b=10;
short s=15;
long l=20;
i=i+1;
b=(byte)(b+1); //evaluates to an integer
s=(short)(s+1); //evaluates to an integer
i=i+1;
System.out.println("i: "+i+" b: "+b+" s: "+s+" i: "+l);
}
```

Issues with Explicit Type

- Loss of information.

  If a float is converted to int, the decimal part is lost.

```
int a;
float b =8.923;
a=b;
//a value will be 8
```

If the Target type has smaller range, Value is reduced modulo the target type's range

```
int i = 257;
        byte b = (byte) i;
```

(Byte's range)

```
//b will store 257%256=1
```

Automatic Type Promotion

- Can also occur in expressions

- Example-

```
byte x = 40;
byte y = 50;
byte z = 100;
int a = (x * z ) + y;
```

- The result of x * z exceeds the range of byte. Java automatically **promotes** byte (and short) to int while evaluating an expression

- It can however cause compile time errors if incompatible types

Rules of Type promotion

- Byte and short are promoted to int
- If one operand is long, the entire expression becomes long
- If one operand is float the entire expression becomes float
- If one expression is double, the entire expression becomes double

Example: Find the output

```
byte b = 42;
char c = 'a';
short s = 1024;
int i = 50000;
float f = 5.67f;
double d = .1234;
double result = (f * b) + (i / c) -(d * s);
```

- 626.778

LECTURE 5
ARRAYS, CLASSES AND OBJECTS

Course: Object Oriented Programming (CS F213)

Prof. Anita Agrawal,

BITS Pilani-K.K.Birla Goa campus

PREDICT THE OUTPUT OF THIS CODE

```
short num1=20000;
short num2=16000;
Short num3=40000;
byte A1=110;
byte A2=120;
byte sum_byte = A1+A2;
int sum_ short = num1+num2 +num3;
System.out.println("sum_short: "+sum_short+"
sum_byte: "+sum_byte);
```

Output:
Sum_short: ??? Sum_byte: ???

EXAMPLE

```
short num1=20000;
short num2=16000;
Short num3=40000;
byte A1=110;
byte A2=120;
byte sum_byte = A1+A2; //causes an error
int sum_ short = num1+num2 +num3; //
Automatically cast to int
System.out.println("sum_short: "+sum_short+"
sum_byte: "+sum_byte);
```

EXAMPLE

```
int i=2000790000;
int j=1108800000;
long sum = i+j;
System.out.println("sum: "+sum);
```

Sum will be -1185377296 (out of range)

EXAMPLE

```
class CharExample{
public static void main(String arg[]){
char a='A';
char b=66;
int i=67;
char c=(char)i;
int j=a;
a=a+1;
a=(char)(a+1);
a+=1;
a++; System.out.println("a: "+a+" b: "+b+" c: "+c+" j: "+j);
}
}
```

```
class CharExample{
public static void main(String arg[]){
char a='A';
char b=66; // No error: Automatic type conversion of literal to the destination type
int i=67;
char c=(char)i; // Explicit casting is required
int j=a; // Automatic type conversion
// a=a+1; // Error: possible lossy conversion from int to char
a=(char)(a+1); // No error due to explicit cast
a+=1; // No error: Internal casting
a++; // No error: Internal casting
System.out.println("a: "+a+" b: "+b+" c: "+c+" j: "+j);
}
}
```

ARRAYS

- An *array* is a container object that holds a fixed number of values of a single type.
- It allows to store and access large number of values conveniently.
- Syntax: *datatype[] identifier*;
  - Example: int [] age;
  - Alternate form: int age [];
- // this form is discouraged

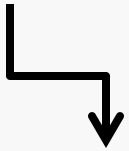
- *dataType* can be a primitive data type like: int, char, Double, byte etc. or an object.

- Identifier (variable name) can be anything provided that it follows the rules and conventions of identifiers.
- int [] age; **[value of age is null]**
- does not actually create an array;
- it simply tells the compiler that this variable will hold an array of the specified type.

CREATING THE ARRAY

- To create an array, use the new operator.

- Syntax: *var-name = new type[size];*
  - Example: age = new int [5];



allocates memory for 5 integer elements and assigns the array to the age variable.

  - Default value: 0 for numeric
  - false for boolean, etc..

ALTERNATE WAY

```
int [] age;
age = new int [5];
```

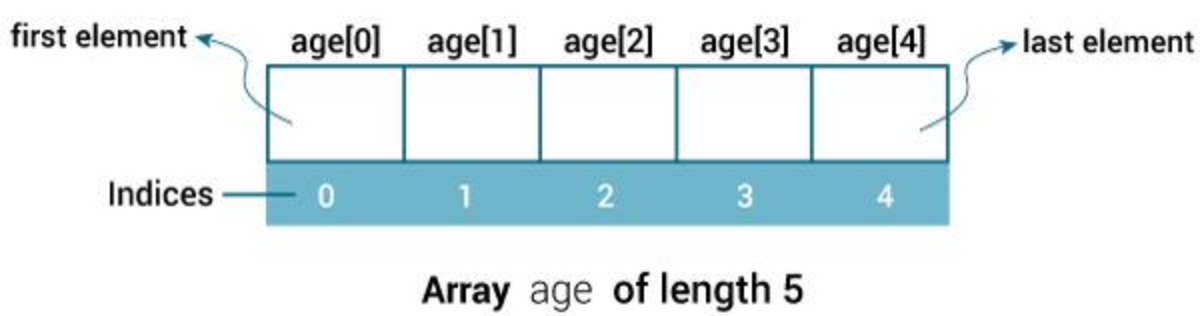
}

```
int[] age = new int[5];
```

Imp: *Once the length of the array is defined, it cannot be changed in the program.*

ACCESSING THE ARRAY

- In case of primitive data types, the actual values are stored in contiguous memory locations.
- Each element is accessed by its numerical *index*.
- Numbering begins with 0.
  - The 5th element, for example, would therefore be accessed at index 4.



INITIALISING THE ARRAY

- int[] age = new int[5];

- age[0] = 10; // initialize first element
- age [1] = 5; // initialize second element
- age [2] = 30; // and so forth

ALTERNATE SYNTAX

- Shortcut syntax to create and initialize an array:
  - int[] age = { 10, 5, 30, 25,40 };
- Here the length of the array is determined by the number of values provided between braces and separated by commas.

ACCESSING THE ARRAY

```
class ArrayExample
{
   public static void main(String[] args)
    {
      int[] age = {12, 4, 5, 2, 5};
      for (int i = 0; i < 5; ++i)
       {
         System.out.println("Element at index " + i
           +": " + age[i]);
       }
     }
 }
```

OUTPUT

- Element at index 0: 12
- Element at index 1: 4
- Element at index 2: 5
- Element at index 3: 2
- Element at index 4: 5

ACCESSING THE ARRAY…..CONTD

```
//Access and alter array elements
class ArrayExample {
   public static void main(String[] args) {
      int[] age = new int[5];
      // insert 14 to third element
      age[2] = 14;
      // insert 34 to first element
      age[0] = 34;

      for (int i = 0; i < 5; ++i) {
         System.out.println("Element at index " + i +": " + age[i]);
      }
   }
}
```

OUTPUT OF THE PROGRAM

```
Element at index 0: 34
Element at index 1: 0
Element at index 2: 14
Element at index 3: 0
Element at index 4: 0
```

MULTIDIMENSIONAL ARRAYS

- An array of arrays/multidimensional array.
- An array whose components are themselves arrays, unlike C or Fortran.
- The rows are allowed to vary in length.

Right index:
column

Left index:
row

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

MULTIDIMENSIONAL ARRAYS (CONTD…)

Example:

```
String[][] names = { {"Mr. ", "Mrs. ", "Ms. "},
{“Manish", “Meeta"}
```

- We can also make use of nested for loops for the initialisation of multidimensional arrays.

MULTIDIMENSIONAL ARRAYS (CONTD…)

Print:

Mr. Manish

Ms. Meeta

```
System.out.println(names[0][0] +names[1][0]);
    // Mr Manish
 System.out.println(names[0][2] + names[1][1]);
   //Ms. Meeta
```

MULTIDIMENSIONAL ARRAYS (CONTD…)

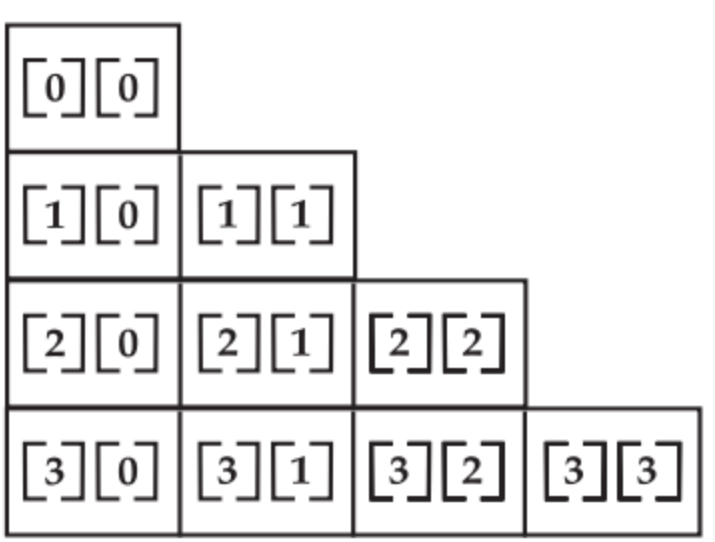
```
class MultiDimArrayDemo
{
    public static void main(String[] args)
     {
         String[][] names =
         {
             {"Mr. ", "Mrs. ", "Ms. "},
             {“Manish", “Meeta"}
         };
        // Mr. Manish
        System.out.println(names[0][0] + names[1][0]);
        // Ms. Meeta
        System.out.println(names[0][2] + names[1][1]);
    }
}
```

MULTIDIMENSIONAL ARRAYS (CONTD…)

- For a multidimensional array, we only need to specify the size for the first (leftmost) dimension
- *int [][] twoD = new int[4][];*
- *twoD[0] = new int[5];*
- *twoD[1] = new int[5];*
- *twoD[2] = new int[5];*
- *twoD[3] = new int[5];*
- *Also possible to write: twoD[0]={10,6,12,13,5};*

MULTIDIMENSIONAL ARRAYS (CONTD…)

- No need to allocate the same number of elements for each dimension
- *int twoD[][] = new int[4][];*
- *twoD[0] = new int[1];*
- *twoD[1] = new int[2];*
- *twoD[2] = new int[3];*
- *twoD[3] = new int[4];*



ALTERNATE DECLARATION

int[] a1, a2, a3;

or

inta1[], a2[], a3[];

Next……classes and objects

Classes & Objects

Lecture 6

Prof. Anita Agrawal

BITS-Pilani, K.K. Birla Goa campus

Classes

- A *class* is the blueprint from which objects are created
- It is an entity that determines how an object will behave and what the object will contain.
- It defines the variables and the methods common to all objects of a certain kind.
- *Syntax:*

```
class classname
{
    members of class
}
```

Class members

- Members of class are:
  - Variables
  - Methods
  - Constructors
- Variables: represent the state of the class.
- Methods: represent behavior of the class.
- Constructors: initialise the state of a new instance of the class

```
modifier class clsName
{
  // instance variable declaration
   type1 varName1 = value1;
   :
   typeN varNameN = valueN;

  //  Constructor
   clsName(cparam1)
   {
      // body of constructor
    }


 // Methods
 rType1 methodName1(mParams1)
  {
    // body of method
   }
   :
 rTypeN methodNameN(mParamsN)
   {
    // body of method
    }
}
```

Variables

Constructor

Methods

Class members

Modifier/ access level

- **Class** access level:
  - Default (accessible only inside the class)
  - Public (accessible in code in any package)
- **Access Level of members: (fields and methods)**
  - Private (only in the class)
  - Package or default (only in this package)
  - Protected (in this package and all sub-classes
  - Public(everywhere)

- A package is a namespace that organizes a set of related classes and interfaces.

Packages

Constructor

- Used for initializing new objects.
- Initializes an object immediately upon creation
- Has the same name as the class in which it resides
- Has optional parameter lists
- Once defined, the constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Has no return type, not even **void**.

```
class rect
 {
     int rlength,rbreadth;
      rect()    //this is a constructor
    {
       rlength=10;
       rbreadth=20;
    }
}
```

Constructor Example

Methods

```
modifier class clsName
{
// instance variable declaration
type1 varName1 = value1;
:
typeN varNameN = valueN;

// Methods
rType1 methodName1(mParams1)
{
// body of method
}
:
rTypeN methodNameN(mParamsN)
{
// body of method
}
}
```

Data: instance variables

code: methods

Methods

- Named methodName1 through methodNameN.
- The return type are rtype1 through rTypeN and
- mParamN are optional parameter lists.

Method Example:

```
int fun_area()
 {
return rlength*rbreadth;
}
```

```
class clsSample
{
   // Variables
     int ctr;
    int i;
    int j;

  // Constructor
    clsSample()
     {
        ctr = 0;
         i = 0;
         j = 0;
      }

// Methods
```

Same name as the class

```
int addition()
  {
    ctr++;
    return i + j;
   }

int NumberofInstances()
   {
      return ctr;
    }
}
```

Objects

- Specimen of the class, a self-contained component consisting of
  - Methods and
  - State

- It determines the behavior of the class.

Objects

- Obtaining objects: two step process.
  - Declare a variable of the class type.
    - does not define an object.
    - simply r*efers* to an object.
  - Acquire an actual, physical copy of the object and assign it to that variable.
    - use the **new** operator.
    - The **new** operator dynamically allocates (that is, allocates at run time) memory for an object and
    - Returns a reference to it.

Syntax

Step 1:

classname object_name

Step 2:

object_name = new classname()

Example:

classname

```
Dog dog1;                //dog1 contains null,
dog1 = new Dog();  // allocates an actual object
```

Shorthand:

```
Dog dog1= new Dog();
```

Each object of class will have its own copy of instance variable: breed, size, color, age

- To access these variables (and methods):
  - Dot(.) operator
- For example:

```
dog1.breed=" german_shepherd ";
dog1.size="Small";
```

OBJECT AND CLASS EXAMPLE: MAIN INSIDE CLASS

```
//Class Declaration
public class Dog
 {
    // Instance Variables
    String breed;
    String size;
    int age;
    String color;

    // method 1
    public String getInfo()
   {
        return ("Breed is: "+breed+" Size is:"+size+" Age is:"+age+" color
is: "+color);
    }

 public static void main(String[ ] args)
 {
        Dog dog1 = new Dog();
       dog1.breed=“german_shepherd";
        dog1.size="Small";
        dog1.age=2;
        dog1.color=“brown";
        System.out.println(dog1.getInfo());
    }
}
```

- Output:

Breed is: german_shepherd Size is:Small Age is:2 color is: brown

Object and Class Example:

```
// Class Declaration
class Dog {
    // Instance Variables
    String breed;
    String size;
    int age;
    String color;

      // method 1
    public String getInfo()
 {
        return ("Breed is: "+breed+" Size is:"+size+" Age is:"+age+" color is:
"+color);
    }
}
public class Execute{
    public static void main(String[] args) {
        Dog dog1 = new Dog();
        dog1.breed=“German_Shepherd";
        dog1.size="Small";
        dog1.age=2;
        dog1.color="white";
        System.out.println (dog1.getInfo());
    }
```

- **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

- **Information-hiding:** By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

- **Code re-use:** If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

Benefits of Objects

- **Pluggability and debugging ease:** If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement.

- Write down a java program to create a class maruti_suzuki which contains the following:

(i) Variables:

model,

type,

mileage,

auto_transmission,

number of airbags.

(ii) One Method named **show_info** of return type String to display all the info.

(iii) Three objects ob1, ob2, ob3

CONSTRUCTORS

LECTURE 7

PROF. ANITA AGRAWAL

BITS, PILANI-K.K.BIRLA GOA CAMPUS

CONSTRUCTORS REVISITED

- **Used for initializing new objects.**
- **Initializes an object immediately upon creation**
- **Has the same name as the class in which it resides**
- **Has optional parameter list**
- **Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes.**
- **Has no return type, not even void.**

THE ‘NEW’ OPERATOR …..REVISITED…….

class_name variable_name = new class_name();

Dog dog1= new Dog();

- **Parentheses after the class name calls the constructor**
- **If no constructor is defined for a class, java creates a default constructor**
- **……Example follows…..**

```
class Dog
  {
   String breed;
   String size;
   int age;
   String color;
  }
 public class Execute
  {
   public static void
    main(String [ ] args)
    {
      Dog dog1 = new Dog();
     }
   }
```

```
class Dog
   {
      String breed;
      String size;
      int age;
      String color;
    }
     Dog()
  {
     Breed =0;
     Size=null;
     Age=0;
     String=null;
   }

   public class Execute
   {
     public static void main(String[]args)
       {
         Dog dog1 = new Dog();
        }
    }
```

GARBAGE COLLECTION

- In Java, objects are dynamically operated by the **new** operator
- Deallocation is done automatically using

  Garbage Collection.
- Garbage collection occurs rarely during program execution

– System.gc();

- When no reference to an object exists,
  - Java assumes that the object is no longer needed

- The memory occupied by object can be reclaimed
- No explicit need to destroy objects

Next…..access control

Access Control and Static Keyword
Lecture 8

Prof. Anita Agrawal,
BITS-Pilani, K.K.Birla Goa campus

Controlling access to members of a class

- Access level modifiers determine whether other classes can use a particular field or invoke a particular method.
- Two levels of access control:
  - At the top level—public, or *package-private* (no explicit modifier).
  - At the member level—public, private, protected, or *package-private* (no explicit modifier).
- Class modifiers:
  - Public: The class is visible to all classes everywhere
  - Package-Private: (default): visible only within its own package

- Class Member modifiers:
  - Public modifier: The member is visible everywhere
  - Package Private: The member is visible only within the package in which it has been defined
  - Protected: The member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package
  - Private: the member can only be accessed in its own class.

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Access levels, why???

- Access levels affect you in two ways.
  - First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use.
  - Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Tips on choosing access level

- If other programmers use your class, you want to ensure that errors from misuse should not happen. Access levels can help you do this.
  - Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
  - Avoid public fields except for constants. Public fields tend to link you to a particular implementation and limit your flexibility in changing your code.

Encapsulation

- Access Control is an important aspect of Encapsulation
- It is the mechanism that binds together code and the data it manipulates

Another Example

```
//Java program to illustrate using class from different class with
private modifier
class Honda
{
private void display()
    {
        System.out.println("Private");
    }
}

class B
{
public static void main(String args[])
    {
        Honda  amaze = new Honda();
    //trying to access private method of another class
         amaze.display();
    }
}
```

Output-

error: display() has private access in Honda

amaze.display();

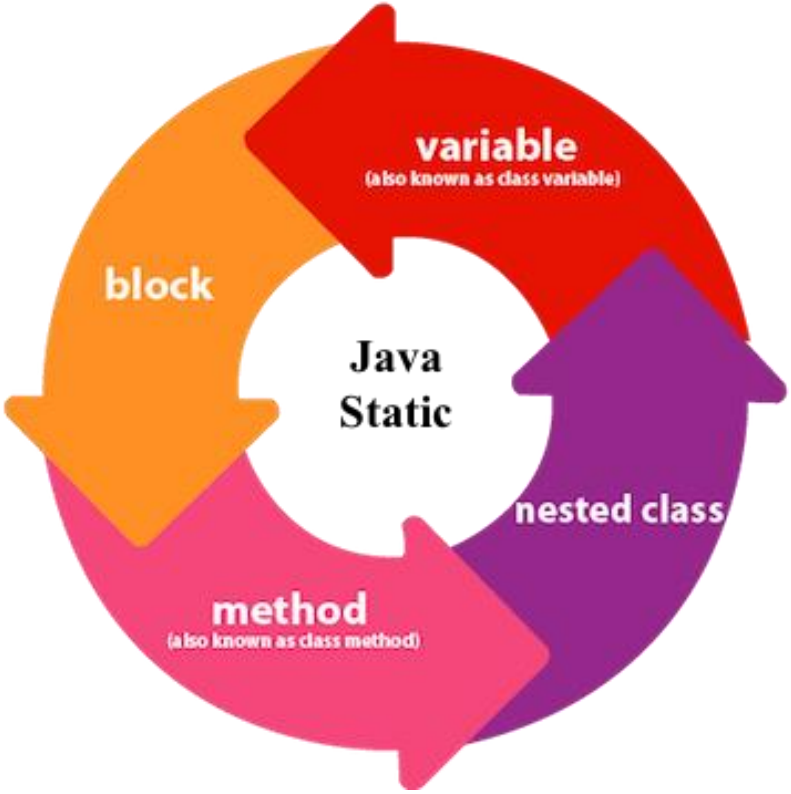
Static modifier

- When a number of objects are created from the same class, they each have their own distinct copies of *instance variables*.
- Each object will have its own values for these variables, stored in different memory locations.
- Sometimes, we want to have variables that are common to all objects.
  - Use *static modifier*
  - Static variables are associated with the class, rather than with any object.
  - Every instance of the class shares the class variable, which is in one fixed location in memory.

Static keyword

- The **static keyword** is used mainly for memory management.

- It is applicable to
  - blocks
  - variables
  - nested classes
  - methods



Static Keyword

- It precedes variable declaration with the keyword static and is common to all objects.

- Ways of intialisation:
  - Directly: modifier(if any) static var_type var_name;

    Example: public static int x =2;

or…….

  - Declare a static block

    Example: static int x;

```
static {
        x = 2;
}
```

- Any object can change the value of the class variable, but class variables can also be manipulated without creating an instance of the class.
  - Syntax: class_name . var_name = value;
  - Example: class A. x = 3;

Static Methods

- Methods can also be declared as static.
  - For Example in public static void main() – main() can be called without creating an object

- Restrictions
  - They can only access static data
  - They can call only other static methods
  - They can't refer to this or super(Inheritance)

OVERLOADING
LECTURE 9

BITS-PILANI,K.K. BIRLA GOA CAMPUS

Prof. Anita Agrawal

Method Overloading???

- In Java, two or more methods can have the same name if ……
  - They differ in parameters:
    - different number of parameters,
    - different types of parameters,
    - Both of the above

Overloaded methods

Method overloading

Example:

```
void func()
{
   ...
}
void func(int a)
{
    ...
}
```

```
void func(double a)
 {
    ...
 }
void func(int a, int b)
 {
    ...
}
```

Example:

```
void func(int a)
{
    ...
}
```

```
int func (int a)
{
    ...
}
```

Overloaded methods may or may not have different return type, but they must differ in parameters they accept.

Without Method Overloading

```
int add2(int x, int y)
{
    return(x+y);
}
int add3(int x, int y,int z)
{
    return(x+y+z);
}
int add4(int w, int x,int y, int z)
{
    return(w+x+y+z);
}
```

With Method Overloading

```
int add(int x, int y)
{
    return(x+y);
}
int add(int x, int y,int z)
{
    return(x+y+z);
}
int add(int w, int x,int y, int z)
{
    return(w+x+y+z);
}
```

Method overloading, why???

- It improves the readability of the program.
- Only one method needs to be invoked

Different ways to perform method overloading…..

- Overloading by changing number of arguments
- Overloading by changing the type of arguments

Important Points

- Two or more methods can have same name inside the same class if they accept different arguments.

- Method overloading is achieved by either:
  - changing the number of arguments.
  - changing the datatype of arguments.

- Method overloading is not possible by changing the return type of methods.

Method Overloading…

- When an overloaded method is called: –
  - Java first looks for an exact match between the arguments and method's parameters
    - Number and type is matched
    - If exact match method is not found
      - Automatic type conversion will take place-> Overload resolution

…over_convert.java

Constructor Overloading

- Like methods, constructors too can be overloaded in a class.

```java
class Data{

    Data()
    {
        System.out.println("Constructor without Parameter");
    }

    Data(int i)
    {
        System.out.println("Constructor with Integer");
    }

    Data(String str)
    {
        System.out.println("Constructor with String");
    }
}

public class Javaapp {

    public static void main(String[] args) {

        Data d1  = new Data();
        Data d2  = new Data(15);
        Data d3  = new Data("String");
    }
}
```

Passing objects as parameters

- Just like primitive types, objects can also be used as parameters to methods.

- Constructors can also have Objects as parameters
- –Useful in duplicating objects

Argument Passing

- Call-by-value: Value of argument passed to the parameter of subroutine

  Changes made to the parameter don't affect the argument.

- Call-by-Reference: Reference to an argument is passed to the parameter of subroutine.

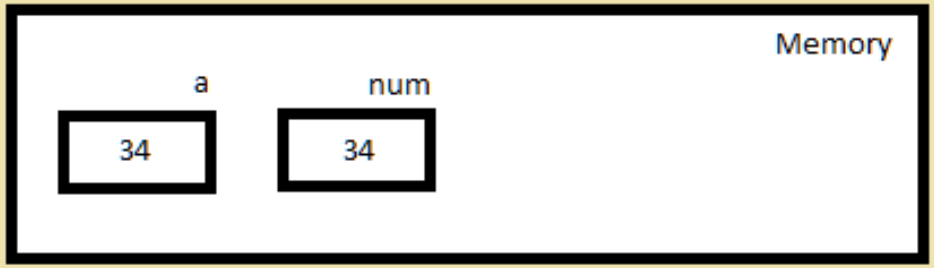
  Changes made to the parameter will affect the argument.

- In Java,
  - Primitive types: Call-by-value
  - Objects: call by reference

Call By Value

public void meth (int num)

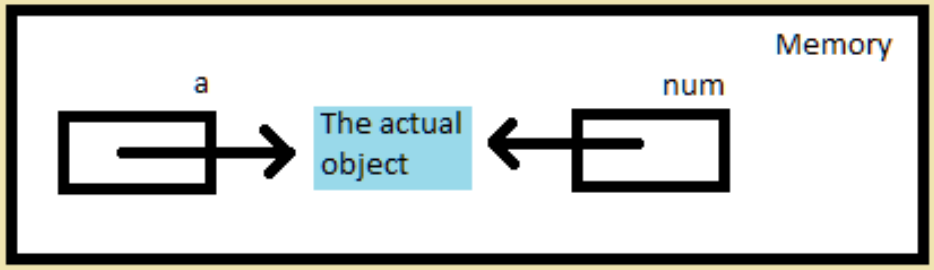
int a = 34;
meth (a)



Call by Reference

public void meth (Number num)

Number a = new Number (34);
meth (a) ;



LECTURE 10A

Overloading

Prof. Anita Agrawal,
BITS, Pilani-K.K.Birla Goa campus

Constructor overloading

- Example: Write a java program for the following:

- Class circle with three variables to hold fractional values for: x,y coordinates of the center and radius r.

- The user may or may not give the values for these.
- If only radius is specified, take origin as the center.
- If nothing is specified, make it a unit circle centered at the origin.
- Further, the class should be capable of computing and returning the values of the circumference and the area

col2...

Java Copy ….

- In Java, there is no operator to create a copy of an object.
- Assignment operator will create a copy of reference variable and not of an object

……..cao

Methods of Copying in Java

- By using constructor
- By assigning the values of one object to another
- By clone() method of Object class

Copying values by using constructor

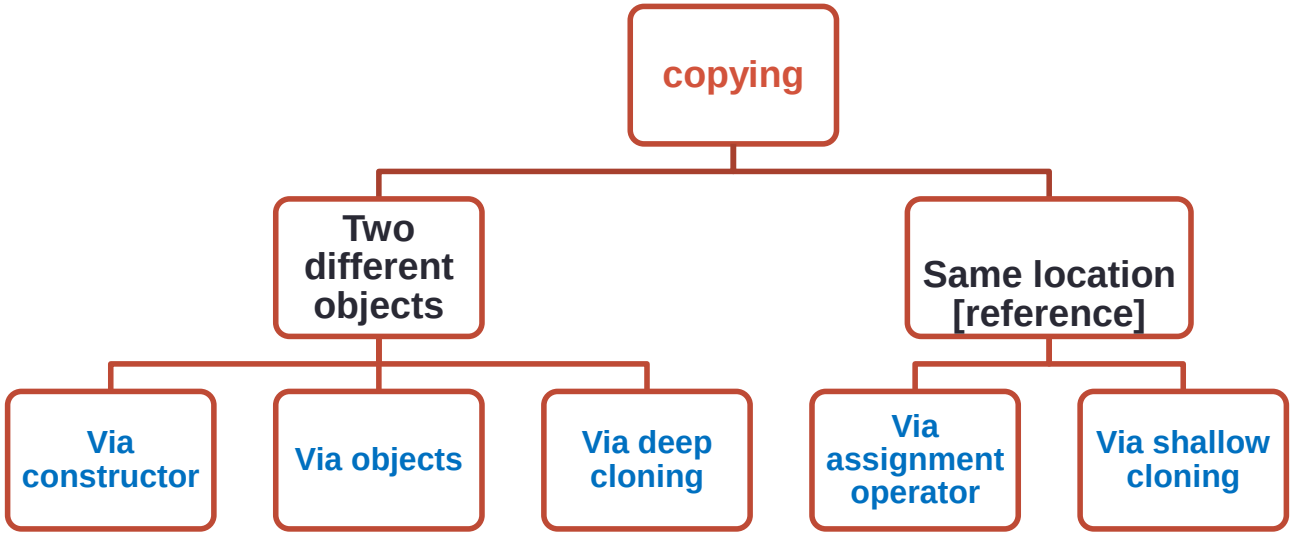
….cbc

Copying by assigning values of one object to another

- The values of one object can be copied into another object by assigning the object's values to another object.

Copying by clone method of object class

Summary of copying methods in Java



Difference between constructor and method in Java

Constructor	Method
Used to initialise the state of an object	Used to expose behavior of an object
Must not have a return type	Must have return type
Is invoked implicitly	Is invoked explicitly
The Java compiler provides a default constructor if you do not have any constructor	Method is not provided by compiler in any case
Name must be same as class name	Name may or may not be same as class name

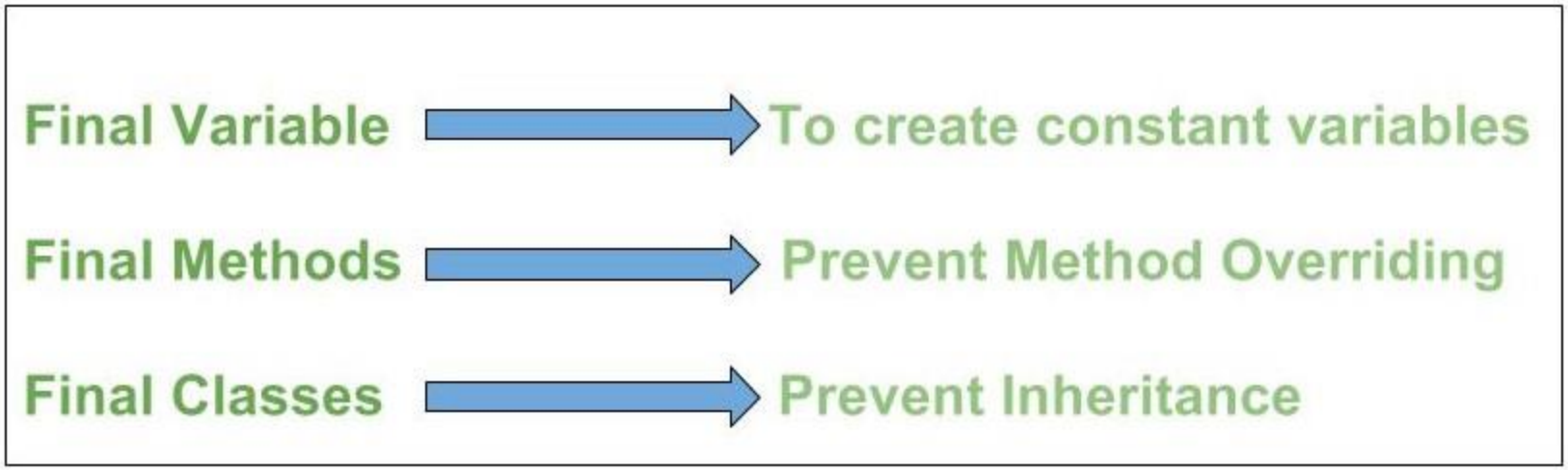
ARGUMENTS & NESTED CLASSES
LECTURE 10B

Prof. Anita Agrawal,

BITS, Pilani-K.K.Birla Goa Campus

final keyword in java

- final is a non-access modifier applicable to:
  - Variable
  - Method
  - Class



- ***final*** variable:
  - When a variable is declared with *final* keyword, it's value cannot be modified.
  - It is essentially a constant.
  - If reference:
    - cannot be re-bound to reference another object
    - Internal state of the object pointed can be changed

- The final variable has to be initialized, otherwise compile-time error
- Can be initialized only once

Ways to initialize the final variable

- **Initialization during declaration:**

Example:

  - final int num = 7;
  - static final double length = 52.5
- If not initialised during declaration: Blank variable

- Blank variable initialisation:
  - inside the instance initialisation block
  - inside the constructor
  - inside all the constructors if more than one constructor
  - inside the static block if a static variable

```
//Java program to demonstrate different
// ways of initializing a final variable
 class A
{
   // a final variable direct initialize
   final double width = 10.5;

   // a blank final variable
   final int sum;

   // another blank final variable
   final int  height;

   // a final static variable PI direct initialize
   static final double PI = 3.14;

   // a  blank final static  variable
   static final double grav_const;
```

```
// instance initializer block for initializing sum
  {
      sum = 25;
  }
// static initializer block for initializing grav_const
  static {
      grav_const = 9.8;
  }

// Constructor for initializing height. Note that if there are more // than one constructors, you must initialize height in them also

public OOP()
  {
      height = 20;
  }
}
```

- As you know that a final variable cannot be re-assigned. But in case of a reference final variable, internal state of the object pointed by that reference variable can be changed.

Note: This is not re-assigning. This property of *final* is called *non-transitivity*.

- The *non-transitivity* property also applies to arrays.,

final arrays.

// Java program to demonstrate reference final variable

finref……

- A final variable cannot be reassigned, doing it, will throw compile-time error.

Java program to demonstrate re-assigning final variable will //throw compile-time error

……finref2

When a final variable is created inside a method/constructor/block, it is called local final variable, and it must be initialized once where it is created.

Java program to demonstrate local final variable

```
class A
{
    public static void main(String args[])
    {
        // local final variable
        final int x;
        x = 15;
        System.out.println(x);
    }
}
```

Difference between a normal variable and a final variable

- Values can be reassigned to normal variables, but final variable values cannot be changed once assigned.
- Use final variables only for the values that are required to remain constant throughout the execution of program.

Final classes

- Declared with *final* keyword.
- A final class cannot be extended(inherited).
- Uses of a final class :
  - To prevent inheritance, as final classes cannot be extended.
  - To create an immutable class like the predefined String class.
    - You can not make a class immutable without making it final.

final method

- Declared with a final keyword.
- A final method cannot be overridden
- Declare methods with final keyword to follow the same implementation throughout all the derived classes.

```
//program to illustrate final keyword with the method
class A
{
    final void m1()
    {
        System.out.println("This is a final method.");
    }
}

class B extends A
{
    void m1()
    {
        // COMPILE-ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

VARIABLE-LENGTH AND COMMAND LINE ARGUMENTS

Lecture 11

BITS Pilani, K.K.Birla Goa Campus

Prof. Anita Agrawal

Command-Line Arguments

- **Command line arguments** is a methodology in which user will give inputs through the console.
- A Java application can accept any number of arguments from the command line.
- Syntax: java class_name command_line_arguments
- JVM wraps the command-line arguments and supplies to args[ ].
  - Can be checked by using args.length

Example

…….cla3

- The space character separates command-line arguments. And hence each word is displayed on a new line by itself.

Variable Arguments (var…args)

- Varargs…(Variable Arguments): A feature in Java that simplifies the creation of methods that need to take a variable number of arguments.

- A method that takes a variable number of arguments is a varargs method.

- An argument that can accept variable number of values is a vararg.

- Syntax:
- A variable-length argument is specified by three periods(…).

Example:

```
public static void fun(int ... a)
 {
    // method body
 }
```

How varargs work behind the scene…

- The **...** syntax tells the Java compiler that the method can be called with zero or more arguments.
- void f1(int …nums)
- As a result, *nums* variable is implicitly declared as an array of type int[ ]. Thus, inside the method, *nums* variable is accessed using the array syntax.
- In case of no args, the length of nums is 0.

Example..

Other parameters

- The varargs method can have other parameters too in addition to varargs.
- But there can exist only one varargs parameter.
- The varargs parameter should be written last in the parameter list of method declaration.

Example:

- int sum(int x, float y, double … z)
  - The first two arguments are matched with the first two parameters and the remaining arguments belong to z

Erroneous varargs examples

- Specifying varargs as the middle parameter of method instead of last one:

int sum (int x, double … z, float y,)

- Specifying two varargs in a single method:

void method(string...abc, int... x)

- Specifying varargs as the first parameter of method instead of last one:

void method(int...x, String abc)

Example: Method using NoVarags

```
class A {

    public int addnum(int a, int b){
        return a+b;
    }

    public int addnum(int a, int b, int c){
        return a+b+c;
    }

    public static void main( String[] args ) {
        A obj = new A();
        System.out.println(obj.addnum(1, 2));
        System.out.println(obj.addnum(1, 2, 3));
    }
}
```

Output:
3
6

- addnum() method had to be overloaded to make it work for 3 arguments.
- What if the user wants to add 5 numbers or 10 or 100?
- This can be handled in a neat way with the use of varargs.

- As you can see, varargs can be really useful in some situations. However, if you are certain about the number of arguments passed to a method, use method overloading instead. For example, if you are certain that sumNumber() method will be used only to calculate the sum of either 2 or 3 arguments, use overloading like in the first example.

Varargs method overloading

- Similar to typical methods, we can overload vararg methods too.

Varargs method… Important Points

- While defining method signature, always keep varargs at the last
- The variable argument must be the last argument passed to the method. Let's consider, you invoked display() method like this:

- display(1, 2, 3, 4);

- And, your display() method is defined as:

- public void display(int ... q, int p){
- // method body
- }

- In this case, compiler cannot figure out the number of arguments passed to q.
- However, if you define your method as:
- public void display(int p, int ... q) {
- // method body
- }
- The Java compiler assigns the first argument to p, and the remaining int arguments are assigned to q.

Ambiguity in varargs method overloading

```
class A {

    static void test(int ... vargs) {
        // method body
    }

    static void test(int n, int ... vargs) {
        // method body
    }
}
```

- In the above program, the compiler gets confused if you try to invoke the test() method even though test() methods are overloaded and accepts different number of arguments.

- The compiler doesn’t know which method to call. The compiler may think, you are trying to call test(int ... vargs) with one varargs argument. Also, the compiler may think, you are trying to call test(int n, int ... vargs) with argument passed to the first parameter with empty second parameter.

- Since there are two possibilities, it causes ambiguity. Because of this, sometimes you may need to use two different method names instead of overloading varargs method.

NESTED CLASSES

LECTURE 12
PROF. ANITA AGRAWAL
BITS PILANI, K.K.BIRLA GOA CAMPUS

- The Java programming language allows you to define a class within another class. Such a class is called a *nested class.*
- This allows for logical grouping of classes that are only used in one place.
- It increases encapsulation.
- Creates more readable and maintainable code.
- *Syntax:*

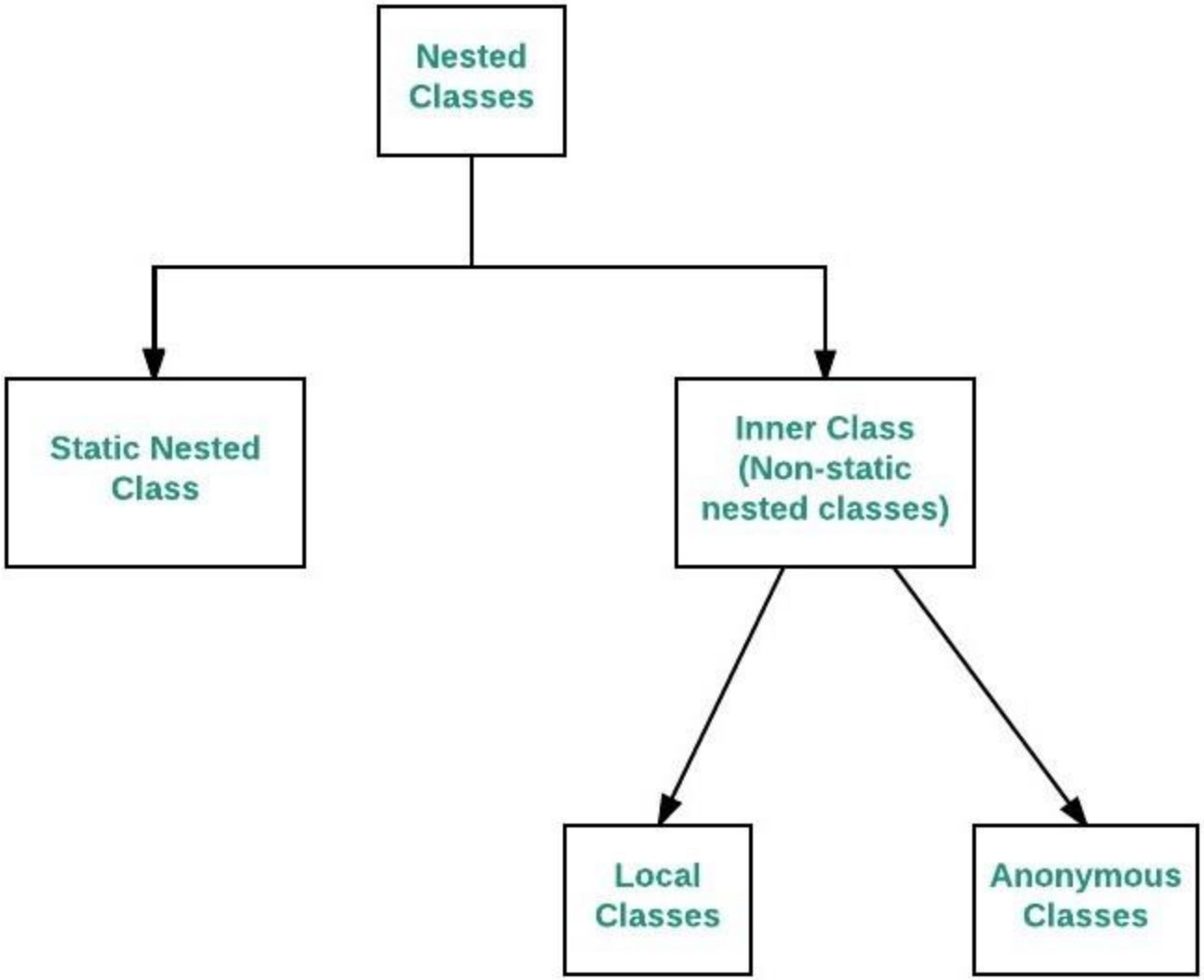
```
class OuterClass
{
 ...
      class NestedClass
   {
         ...
     }
 }
```

CATEGORIES OF NESTED CLASSES

- **NON-STATIC (INNER CLASSES)**

- **STATIC (STATIC NESTED CLASSES)**

- **NON-STATIC (INNER CLASS):**
  - The inner class has access to members of the enclosing class (i.e. outer class).
  - To instantiate an inner class, you must first instantiate the outer class.


Nested Classes
Static Nested Class
Inner Class (Non-static nested classes)
Local Classes
Anonymous Classes

Inheritance

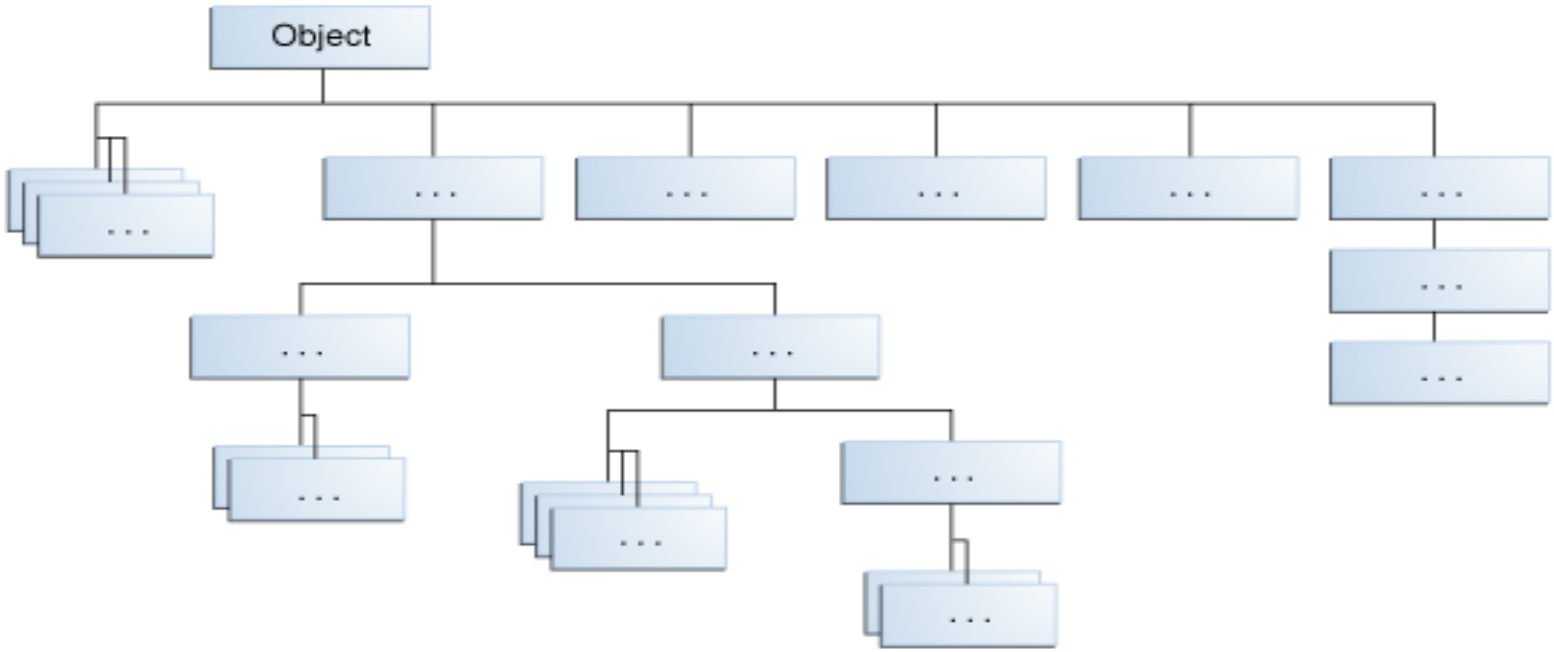
Lecture 13

Prof. Anita Agrawal,
BITS –Pilani, K.K.Birla Goa campus

What is inheritance?

- In Java language, classes can be *derived* from other classes, thereby ***inheriting*** fields and methods from those classes.
- A class that is derived from another class: a ***subclass/derived class/extended class/ child class***).
- A class from which it is derived: ***superclass/base class/parent class***

- Except for the class Object, every other class has one and only one direct superclass (**single inheritance**).
- If there is no explicit superclass for a class, it is implicitly a subclass of the class object.
- Classes can be derived from classes which are derived from classes, ……. And are ultimately derived from class object.


Object

- A subclass inherits all the *members** (**fields, methods, and nested classes**) from its superclass.
- Constructors of the superclass are not inherited by subclass, but can be invoked from the subclass.

- A subclass inherits all of the ***public*** and ***protected*** members of its parent, no matter what package the subclass is in.
- If the subclass is in the same package as its parent, it also inherits the ***package-private members*** of the parent.
- A subclass does not inherit the private members of its parent class.
- However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

Fields in subclasses

- You can use the inherited members as is, replace them, hide them, or supplement them with new members.
- The inherited fields can be used directly, just like any other fields. And can also change their values (replacing)
- You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (**not recommended**).
- You can declare new fields in the subclass that are not in the superclass.

Methods in subclasses

- The inherited methods can be used directly as they are.
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus ***overriding*** it.

- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword **super.**

Overriding

- Inheritance1

super

- Inheritance2
- inheritance3

Inheritance

Lecture 13 -B

Prof. Anita Agrawal,
BITS –Pilani, K.K.Birla Goa campus

Using super keyword

Super has two forms:

- To call the superclass' constructor
- To refer members of the superclass
  - Hidden members of the superclass

Using super with constructor

- A subclass can call superclass constructor

  super(parameter-list);

- *super()* must be the **first** statement executed inside subclass' constructor

Using super with Constructor

- If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the “no-argument” constructor of the superclass.

- If the super class does not have a “no-argument” constructor, you will get a compile-time error.

- If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, there will be a whole chain of constructors called, all the way back to the constructor of Object.

constructor chaining

- **super( )** can be called using any form defined by the superclass
- The constructor executed will be the one that **matches the arguments**
- Even the object of subclass can be passed to the constructor of superclass

Access to superclass members by using super keyword

- Used to access members of superclass
- General form

  super.memberVar

  super.memberMethod()

- Applicable when member of subclass hides the members of superclass

- Same name variables or methods

….example

How to refer a subclass object

- **Two approaches:**
  - **Referencing using Superclass reference:** A reference variable of a superclass can be used to refer any subclass object derived from that superclass.
  - **Referencing using subclass reference:** A subclass reference can be used to refer its object.

subclassobj

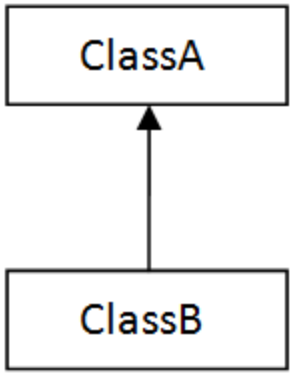
Inheritance

Lecture 14

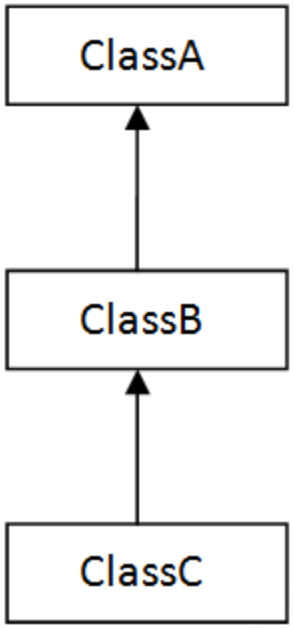
Prof. Anita Agrawal,
BITS –Pilani, K.K.Birla Goa campus

Types of inheritances in Java

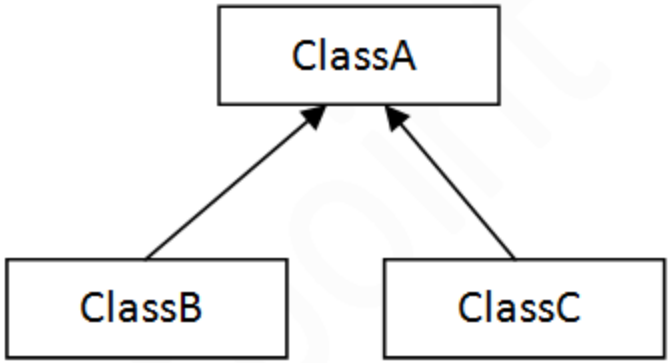
- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance (Through Interfaces)



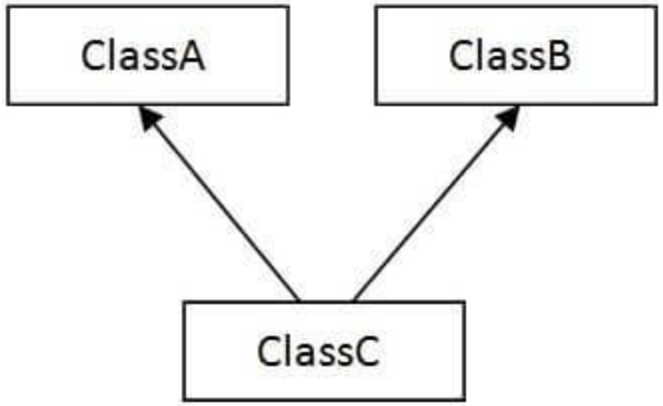
1) Single



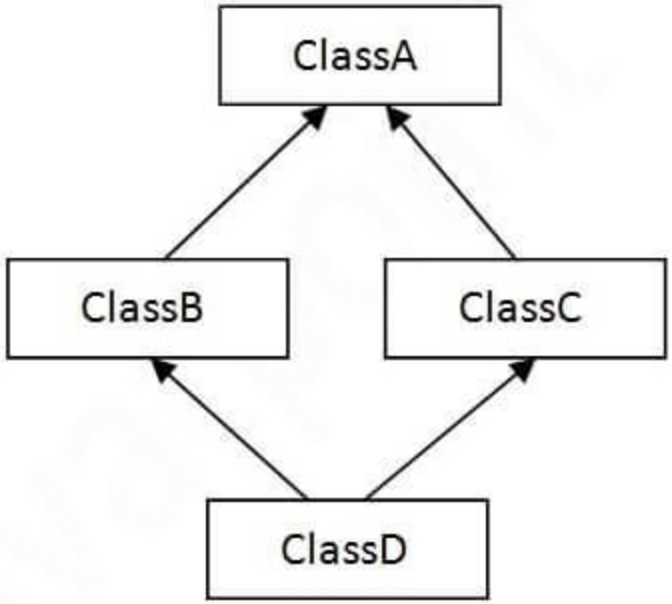
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

- **Single Inheritance**: One class (sub class) extends another class (superclass)

(one class only)

- **Multilevel Inheritance**: In other words, a derived class will be inheriting a base class and the derived class will act as the base class to other class.

- **Multiple Inheritance**: One class extends more than one class. Java does not support multiple inheritance through classes. It can support this only through interfaces

- **Hierarchical inheritance**: One class is extended by many subclasses.
- **Hybrid Inheritance**: It is a combination of single and multiple inheritance.

Summary:

- In Java, when an "Is-A" relationship exists between two classes, we use Inheritance.
- The keyword '**extend**' is used by the subclass to inherit the features of the superclass.
- Inheritance is important since it leads to **reusability** of the code.

Method overriding

- Overriding occurs *only* when the names and the type signatures of the two methods in the subclass and the superclass are identical.

Dynamic method dispatch

- A mechanism by which a call to an overridden method is resolved at run time, rather than compile time.
  - **Is one of the ways in which java supports run time polymorphism**
- A superclass reference variable can refer to a subclass object. -**upcasting**.
- Java uses this fact to resolve calls to overridden methods at run time.

- If a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

- In Java, we can override methods only, not the variables(data members), so **runtime polymorphism cannot be achieved by data members.**

Final revisited ….

Final classes:

- When a class is declared with ***final*** keyword, it is called a **final** class.
- A final class **cannot be extended(inherited)**.
- Uses of a final class :
  - To prevent inheritance
    - Example: All Wrapper classes like Integer, Float etc. are final classes.
  - To create an immutable class
    - Example: The predefined **String** class.

  You can not make a class immutable without making it final.

Finaldemo.java

Overloading vs. Overriding:

Overriding occurs when both, the **name** and the **type signature** of the two methods are **identical.** Otherwise, the two methods are simply overloaded

Final method

- Declared with a final keyword.
- A final method cannot be overridden by subclasses.
  - Example: the Object class.
- Declare methods with final keyword to follow the same implementation throughout all the derived classes.

Finalmethoddemo.java

- During inheritance, declare methods with final keyword for which we require to follow the same implementation throughout all the derived classes.

- It is not necessary to declare final methods in the initial stage of inheritance(base class always).

- final method can be declared in any subclass for which we want any other class extending this subclass, must follow same implementation of the method as in the that subclass.

- Since private methods are inaccessible, they are implicitly final in Java. So adding *final* specifier to a private method doesn't add any value. It may in-fact cause unnecessary confusion.

What happens if a class is declared final?

- Declaring a class as final implicitly declares all of its methods as final, too.

Abstract Classes

Lecture 15

Prof. Anita Agrawal,

BITS,Pilani-K.K.Birla Goa Campus

Abstract classes, why and when????

- When it is difficult or often unnecessary to implement all the methods in parent class:
  - *Make the parent class an abstract class
    - *A special class, Not complete on its own.
  - *Make an extended class
    - * Implement the method there
- Syntax: abstract class_name

Abstract classes and instantiation..

- Abstract class cannot be instantiated

- To use this class, you need to use the object of the extended class.

Extended class…Requirements

- A class derived from the abstract class must implement all those methods that are declared as abstract in the parent class.

Abstract method

– A method which is declared as abstract and does not have implementation is known as an abstract method.

– Syntax for abstract method:

abstract void method_name();
//no method body

- Use the object of the child class to call all methods of parent class.

- **Note:** If a child does not implement all the abstract methods of abstract parent class, then the child class must need to be declared abstract as well.

More about Abstract class…

- It cannot be instantiated, but we can have references of abstract class
- It may include abstract methods and normal methods

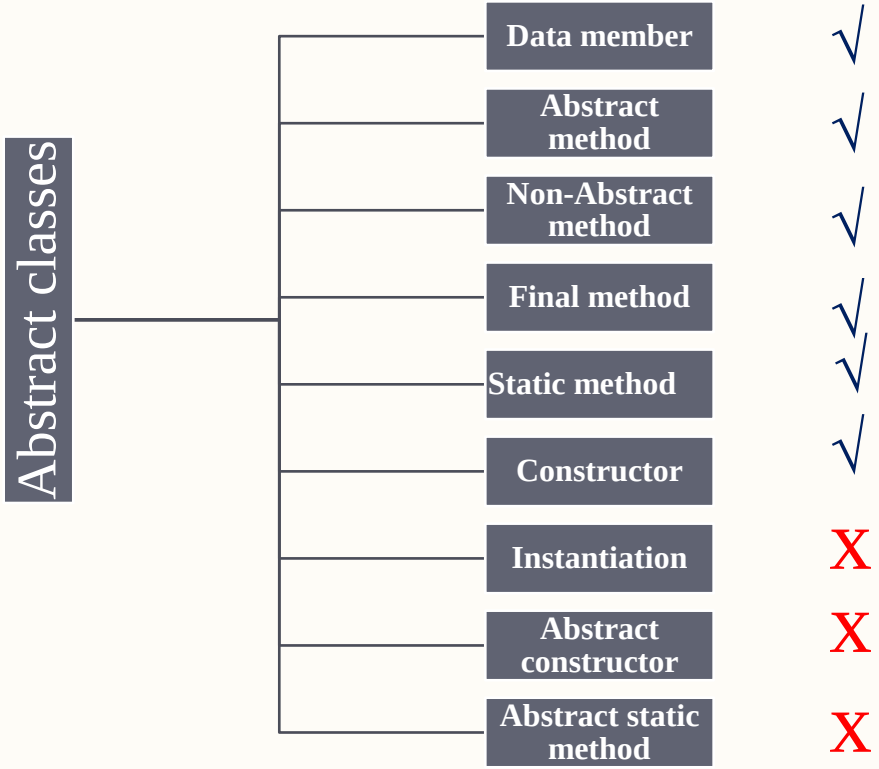
- An abstract class can be abstract even without any abstract method.

- Advantage: This allows us to create classes that cannot be instantiated, but can only be inherited.

- Abstract classes can also have final methods
- An abstract class can contain constructors.
- And a constructor of abstract class is called when an instance of a inherited class is created.
- An abstract class can have static methods
- Static methods cannot be abstract

– Constructors cannot be abstract

In a nutshell, abstract classes can……..



Examples

Interfaces Lecture 16

Prof. Anita Agrawal

BITS- Pilani,K.K.Birla Goa campus

What is an interface?

- Is a core part of java programming.
- It is a reference type in Java.
- It is similar to class.
- It is a collection of abstract methods only???.

Why Interfaces?

- To achieve **total abstraction**.
- To define the **contract** for the subclasses to implement.
- To support **multiple inheritance**.

Interfaces can also contain: (Java 8/9)

- Only constants
- Default methods/defender/virtual extension (consists of body)
- Static methods (consists of body)

Differences between interface and class

- **Interfaces cannot be instantiated.**
- **Classes can be instantiated.**

- **Interfaces cannot contain constructors**
- **Classes contain constructors**

- **Interfaces are used to achieve multiple inheritance**
- **Classes cannot be used to achieve multiple inheritance**

- **Interfaces cannot be declared as private, protected. They are by default public.**
- **Classes can be.**

Differences between Abstract class and Interface

- Variables in interface are by default public, static and final.

An abstract class may contain variables which are final, non-final, static, nonstatic, private, protected etc.

- A Java interface can be implemented using keyword “implements

Abstract class can be extended using keyword “extends”

- Interface can’t provide the implementation of abstract class

Abstract class can provide the implementation of interface.

- Interface can have only abstract methods. From Java 8, it can have default and static methods also.

Abstract class can have abstract and non-abstract methods.

- An interface can extend another Java interface only, it cannot extend any class.

An abstract class can extend another class and implement multiple interfaces.

- While providing implementation of any method in class of an interface, it needs to be mentioned as public.

This is not the requirement in Abstract classes

Some more properties of Interfaces

- An interface is implicitly abstract. We need not explicitly declare it as such.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.

Implementing Interfaces

- A class formally implements an interface by
  - stating so in the class header in the implements clause
  - a class can implement multiple interfaces: the interfaces are listed in the implements clause, separated by commas

- If a class asserts that it implements an interface, it must define all methods in the interface or the compiler will produce errors

Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.

- Multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class

Extending interfaces

- An interface can extend another interface in the same way that a class can extend another class.
- **Implements** keyword is used when a class extends an interface and **extend** keyword is used when an interface extends an interface
- The child interface/class inherits the methods of the parent interface.
- An interface can extend more than one parent interface and the class can extend more than one parent interface too.(multiple inheritance)
- The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

Interface Hierarchies

- Inheritance can be applied to interfaces as well as classes
- One interface can be used as the parent of another
- The child interface inherits all abstract methods of the parent
- A class implementing the child interface must define all methods from both the parent and child interfaces
- Note that class hierarchies and interface hierarchies are distinct (they do not overlap)

Nested Interface

- An interface can be declared inside another interface also.
- We mention the interface as i_**name1.i_name2**

  i_name1 is the name of the interface in which it is nested and

  i_name2 is the name of the interface to be implemented.

Static methods in interfaces

- **Static Methods** in **Interface** are those methods, which are defined in the interface with the keyword static.
- The static method in an interface can be defined in the interface, but these methods cannot be overridden in Implementation Classes.
- To use a static method, Interface name should be instantiated with it, as it is a part of the Interface only.

Default methods

- Java 8 introduces the “Default Method” or (Defender methods) feature, which allows the developer to add new methods to the interfaces without breaking their existing implementation
- ***Default methods*** have introduced as a mechanism to extend interfaces in a backward-compatible way
- ***Default methods*** can be provided to an interface without affecting implementing classes as it includes an implementation

Wrappers Lecture 17

Prof. Anita Agrawal

BITS- Pilani,K.K.Birla Goa campus

Wrappers, what and why???

- Java uses primitive types (also called as simple types) such as int, double etc..to hold the basic data types supported by the language.
- Primitives are not objects.
- Primitives yield better performance, by cutting down the overhead required if making objects of these even for the simplest of calculations.

Wrapper class, what and why??

- While storing in data structures which support only objects (ex. Array list and vectors), it is required to convert the primitive type to object first.

- Objects are needed if we wish to use the reference way of calling a method (because primitive types are passed by value).

- The classes in java.util package handle only objects
- ……
- Java handles this by using a wrapper class.
- A Wrapper class is a class which encapsulates a primitive type within an object.

- When we create an object of a wrapper class, it contains a field and in this field, we can store a primitive data type. In other words, we can wrap a primitive value into a wrapper class object.

- The eight classes of *java.lang* package are known as wrapper classes in java.

The eight classes of *java.lang* package are known as wrapper classes in java.

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Mechanisms

- Two mechanisms are possible:
- Boxing
- Unboxing
- Boxing: conversion of primitive into an object/ The process of encapsulating a primitive value within an object
- Unboxing: conversion of object into primitive/The process of extracting a primitive value from a type wrapper

Wrapper class

- Every wrapper class has two constructors,
  - First constructor takes corresponding primitive data as an argument
  - Second constructor takes string as an argument.
- The string passed to second constructor should be parse-able to number, otherwise you will get run time NumberFormatException.
- Wrapper Class Character has only one constructor which takes char type as an argument.
  - It doesn’t have a constructor which takes String as an argument
  - Because, String can not be converted into Character.
- Wrapper class Float has three constructors
  - The third constructor takes double type as an argument.

Constructors

- **Character constructor….**
  - **Character** is a wrapper around a **char**
- The constructor for **Character** is

Character(char ch)

Specifies the character that will be wrapped by character object being created

Boolean constructors

- **Boolean** is a wrapper around **boolean** values
- It defines the following constructors:
  - Boolean(boolean boolValue) // boolValue must be either true or false
  - Boolean(String boolString) // If *boolString* contains the string “true” (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

Integer constructors…

- int value
- Integer(int num)

- String value
  - Integer(String str)
    - If *str* does not contain a valid numeric value, then a **Number Format Exception** is thrown.

Boxing and Unboxing: Method 1

- Example: Wrappers.java

Numeric type Wrappers

- The most commonly used wrappers are the numeric type.
- All of the numeric type wrappers inherit the abstract class **Number**
- **Number** declares methods that returns the primitive value of an object in each of the different number formats
- These methods are implemented by each of the numeric type wrappers

byte byteValue( )

double doubleValue( )

float floatValue( )

int intValue( )

long longValue( )

short shortValue( )

Boxing and Unboxing (Method 2)

- To obtain the int value contained in an Integer object

  int intValue( ) ;

//Returns the encapsulated int value

- To obtain the **char** value contained in a **Character** object

  char charValue( ) ;

//Returns the encapsulated character

- To obtain a boolean value from a **Boolean** object
  - boolean booleanValue( )

etc…

Boxing and Unboxing

- All of the type wrappers override **toString( )**
- Return the human-readable form of the value contained within the wrapper

  Output the value by directly passing a type wrapper object to println( ) without converting into its primitive type

- Integer x= new Integer(15); //boxing
- int i = x.intValue(); //unboxing
- System.out.println(i+ “ “ + x); // displays 15 15

Autobox and Autounbox(Method 3)

- From J2SE 5.0, two additional features were introduced:

(i) Autoboxing

(ii) AutoUnboxing

(i) Autoboxing: A primitive type is **automatically** encapsulated (boxed) into its equivalent type wrapper (whenever an object is needed)

- No longer necessary to manually construct an object

(ii) Auto-Unboxing: The value of a boxed object is **automatically** extracted (unboxed) from a type wrapper (when its value is needed)

- No need to call a method such as **intValue( )** or **doubleValue()** etc..

Autoboxing

- You need only assign that value to a type-wrapper reference
- Java automatically constructs the object for you
- Example:
- class AutoBox {

```
    public static void main(String args[ ]) {
      Integer j= 15;                    //autoboxing
      int i= j;                         //autoUnboxing
    System.out.println(i+ " " + j);     //Displays 15  15
}
}
```

Autoboxing and methods

- Autoboxing/Auto-unboxing can also occur when
  - An argument is passed to a method
  - When a value is returned by a method

Example:

```
class AutoBox2 {
        static int mat(Integer v) {
                return v ;                          // autounbox
                        }
      public static void main(String args[]) {
                Integer j1= mat (15);               // autobox
                System.out.println(j1);
                                }                   // 15
                        }
```

AutoBoxing/AutoUnboxing in Expressions

- In general, auto-boxing/unboxing take place whenever a conversion into an object or from an object is required.
- Example:
- Integer i1, i2;

```
  i1= 90;              //Autoboxing
  ++i1;                //Auto-unboxing and then reboxing
i2 = i1 + (i1/ 3);     //Auto-unboxing and then reboxing
int i = i1+ (i1/ 3);   //Auto-unboxing and NO reboxing
```

AutoBoxing/AutoUnboxing in Expressions

- Auto-unboxing also allows you to mix different types of numeric objects in an expression
- Once the values are unboxed, the standard **type promotions** and conversions are applied
- You can even use **Integer** numeric objects to control a **switch** statement

AutoBoxing/AutoUnboxing in Expressions

```
Integer i1= 18;
Double d1= 50.5;
d1= d1+ i1;

Integer i2= 2;
switch(i2) {
case 1: System.out.println("one");
break;
case 2: System.out.println("two");
break;
default: System.out.println("error");}
```

Autoboxing/AutoUnboxing Boolean and character values

```
Boolean b = true;                    //Autobox a boolean
if(b)
{
System.out.println("b is true");
 }
Character ch= 'x';                   // box a char
char ch2 = ch;                               // unbox a char
```

When to use objects…

- Should we use objects such as **Integer** or **Double** exclusively, abandoning primitives altogether?
- Example:

```
Double a, b, c;
a = 10.0;
b = 4.0;
c = Math.sqrt(a*a + b*b);
System.out.println("Hypotenuse is " + c);
```

- Far less efficient than the equivalent code written using the primitive type **double**
- Restrict the use of the type wrappers **to only** those cases in which an object representation of a primitive type is required

LECTURE 18

Prof. Anita Agrawal,
BITS-Pilani, K.K.Birla Goa Campus

Package???

- Uptill now, in all the examples, a unique name was given to each class to avoid collision, as all the names were taken from the same name space.
- Java provides a mechanism for partitioning the class name space into more manageable chunks.

..... “**package**”…….

- Additionally, you can define classes inside a package that are not accessible by code outside that package.
- You can also define class members that are exposed only to other members of the same package.

Package

- Package is namespace that organizes a set of related classes and interfaces.
- A mechanism to encapsulate a group of classes, sub classes and interfaces

in short…

A naming and a visibility control mechanism

Benefits of Package

- Preventing naming conflicts:
  - For example there can be two classes with name city in two packages: car.honda.sedan.city and car.honda.auto.city
- Encapsulation
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier

Defining a package

- To create a package,
  - Include the ‘package’ command as the first name in a Java source file.

    syntax: package package_name;
  - Any classes declared within that file will belong to the specified package
- If you omit the **package** statement,
  - The classes are put into the default package, which has no name

Most of the time it is best to define a package for your code.

Package

- Java uses file system directories to store packages
- The **.class** files for the classes belonging to a package must be stored in a directory whose name is same as the package
- The directory name must match the package name exactly
- More than one file can include the same **package**
- The **package** statement simply specifies to which package the classes defined in a file belong.

Package Hierarchy

- You can create a hierarchy of packages
- By simply separating each package name from the one above it by use of a period
  - package *pkg1*[.*pkg2*[.*pkg3*]];
- Need to be stored in pkg1\pkg2\pkg3 in a Windows environment

Compilation and Execution

- The easiest way: Simply create the package directories below your current development directory.
- Put the **.class** files into the appropriate directories (packages)
- Then execute the programs from the development directory

Access Protection

- Classes and packages: Means of encapsulating, Acts as containers
- Classes act as containers for data and code
- Packages act as containers for classes and othersubordinate packages
- Four categories of visibility for class members:
- Subclasses in the same package
- Non-subclasses in the same package
- Classes that are neither in the same package nor subclasses
- Subclasses in different packages

Access Protection

- Anything declared **public** can be accessed from anywhere
- Anything declared **private** cannot be seen outside of its class
- When a member does not have an **explicit access** specification, it is visible to subclasses as well as to other classes in the same package. This is the default access
- If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly
  - Declare that element **protected**

Class Member Access

	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

Import statement

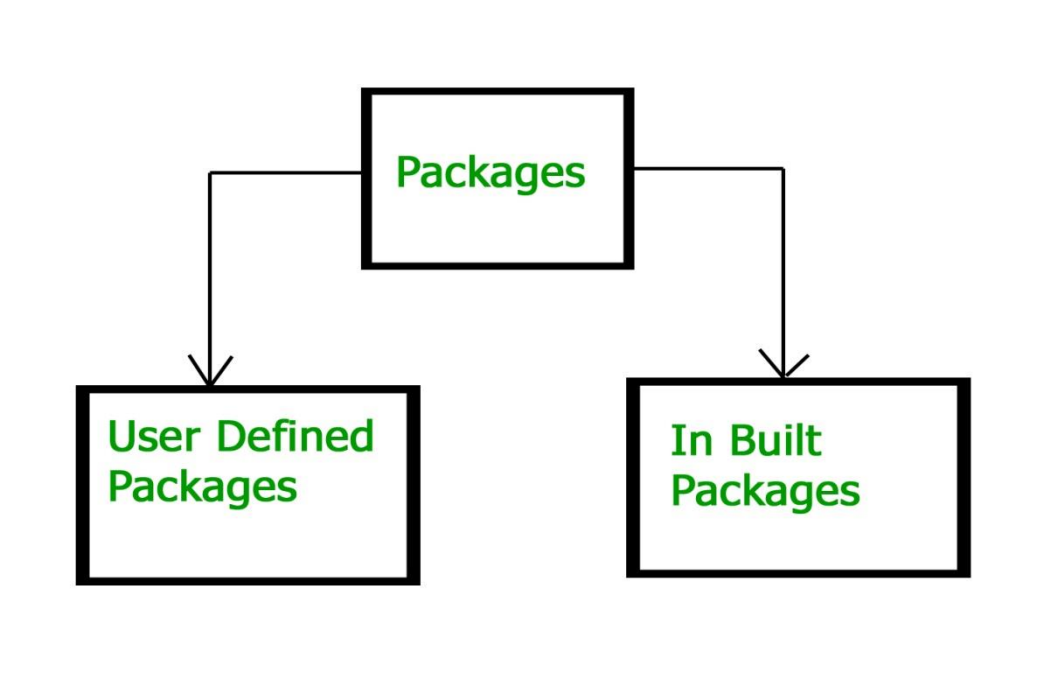
- Packages are a good mechanism for Compartmentalizing diverse classes
- All of the built-in Java classes are stored in packages
- All of the standard classes are stored in some named package
- Classes within packages must be fully qualified withtheir package name
- Tedious to type in the long dot-separated package path name for every class you want to use **import** statement
- Bring certain classes, or entire packages, into visibility

- **import** statements occur
  - immediately following the **package** statement (if it exists)
  - And before any class definitions
- This is the general form of the **import** statement:
  - import *pkg1* [.*pkg2*].(*classname* | *);

Import and public

- When a package is imported
  - Items within the package declared as **public** will be available to non-subclasses in the importing code

Types of Packages



Built-in Packages

- These packages consist of a large number of classes which are a part of Java **API**.
- Some of the commonly used built-in packages are:
  - ✓ **java.lang:** Contains language support classes(e.g classes which define primitive data types, math operations). This package is automatically imported.
  - ✓ **java.io:** Contains classes for supporting input / output operations.
  - ✓ **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
  - ✓ **java.applet:** Contains classes for creating Applets.
  - ✓ **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , menus etc).
  - ✓ **java.net:** Contain classes for supporting networking operations.

Java Lang package

- **Boolean**: The Boolean class wraps a value of the primitive type boolean in an object.
- **Byte**: The Byte class wraps a value of primitive type byte in an object.
- **Double**: The Double class wraps a value of the primitive type double in an object
- **Float:** The Float class wraps a value of primitive type float in an object.
- **Integer** :The Integer class wraps a value of the primitive type int in an object.
- **Long**: The Long class wraps a value of the primitive type long in an object.

User-defined Packages

- These are the packages defined by the user.
- Using static import feature:
  - Static import is a feature introduced in **Java** programming language (versions 5 and above) that allows members (fields and methods) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.

REMAINING TOPICS FROM WRAPPERS

26/03/2019

Deprecated Constructors

- Till now, we have used various constructors for creating object of type wrappers
- Character(char ch)
- Integer(int num)
- Integer(String str)
- These constructors are now **deprecated**
- Discouraged from using these constructors
- Better alternative exists

Utility methods

- ValueOf() method
- Parse method

Utility method ‘valueOf()

- Three types:
  - Wrapper valueOf(String s):
    - Every wrapper class except Character class contains a static valueOf() method to create Wrapper class object for given String.
  - Wrapper valueOf(String s, int radix)
    - Every Integer Wrapper class (Byte, Short, Integer, Long) contains the following method to create a Wrapper object for the given String with specified radix. The range of the radix is 2 to 36.
  - Wrapper valueOf(primitive p) :
    - Every Wrapper class including Character class contains the following method to create a Wrapper object for the given primitive type.

valueOf() for string

```
class integ {
    public static void main(String[] args)
    {
        Integer I = Integer.valueOf(“25");
        System.out.println(I);
        Double D = Double.valueOf(“25.0");
        System.out.println(D);
        Boolean B = Boolean.valueOf("true");
        System.out.println(B);
}
}
```

- Integer I1 = Integer.valueOf(“one"); Will give an error.

Output:25

25.0

true

Wrapper valueOf(String s, int radix)

```
class radi {
    public static void main(String[] args)
    {
            Integer I = Integer.valueOf("1110", 2); //1110 is expressed in binary,
                                                    convert to decimal
        System.out.println(I);
        Integer I1 = Integer.valueOf("1110", 3);  //1110  is expressed in base 3.
                                                  convert it into decimal
        System.out.println(I1);
    }
}
```

Output:

14

39

Wrapper valueOf(primitive p)

```
class cha {
    public static void main(String[] args)
    {
        Integer I = Integer.valueOf(25);
        Double D = Double.valueOf(25.0);
        Character C = Character.valueOf('a');
        System.out.println(I);
        System.out.println(D);
        System.out.println(C);
    }
}
```

Utility method of wrapper class: Parse

parseXxx() method to convert String to primitive.

Two types:

primitive parseXxx(String s) : Every Wrapper class except character class contains the following parseXxx() method to find primitive for the given String object.

parseXxx(String s, int radix) : Every Integer type Wrapper class (Byte, Short, Integer, Long) contains the following parseXxx() method to convert specified radix String to primitive.

Converting string to object via parse…

Example : parse

EXCEPTION HANDLING

Prof. Anita Agrawal,
BITS Pilani, K.K. Birla Goa campus

EXCEPTION HANDLING: AN INTRODUCTION

- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e. at run time

- When an **Exception** occurs, the normal flow of the program is disrupted and the program/Application terminates abnormally,
  - This is not recommended, therefore, these exceptions are to be handled.

WHEN CAN AN EXCEPTION OCCUR???

- Some of the errors are caused by users, some by programmers, others by physical resources that have failed in some manner
- The user has entered invalid data
- A file that needs to be opened cannot be found
- A network connection has been lost in the middle of communication.
- The JVM has run out of memory

TYPES OF EXCEPTIONS

- Checked Exceptions (Compile time Exceptions):
  - An exception that is checked (notified) by the compiler at compilation-time. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

- Unchecked Exceptions (Run-Time Exceptions):
  - An exception that occurs at run-time.
  - Runtime exceptions are ignored at the time of compilation

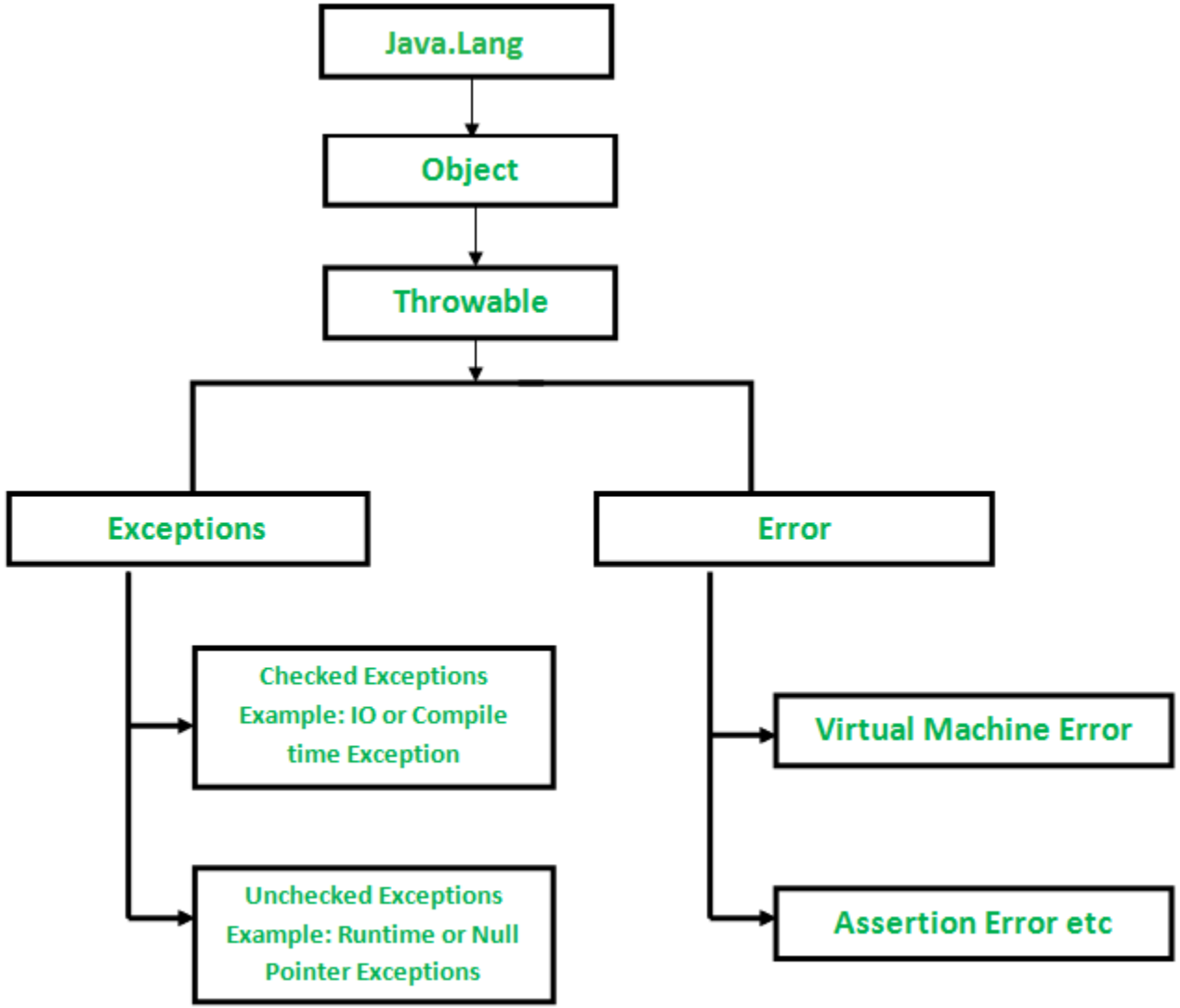
Examples: checked
unchecked

- **Not all computer languages support exception handling**
- Java supports Exception handling.
- It does so by bringing run-time error management into the object-oriented world.
- In Java, Exception is an object.
  - Describes an exceptional condition occurred in a piece of code

EXCEPTION HIERARCHY

- All exception classes are subtypes of the java.lang.Exception class
- The exception class is a subclass of the Throwable class.
- Throwable class also has another subclass called Error.

EXCEPTION CLASS



ERRORS

- Class Error defines the exceptions that are not expected to be caught by your program
- Are abnormal conditions that happen in case of severe failures such as catostrophic failures.
- These are not handled by the Java programs.
- **Errors** are used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE).
  - Example StackOverflow Error .

We will not be dealing with exceptions of type **Error**

ERROR TYPES

- Syntax errors arise because the rules of the language have not been followed. They are detected by the compiler
- Runtime errors occur while the program is running if the environment detects an operation that is impossible to carry out
- Logic errors occur when a program doesn't perform the way it was intended to.

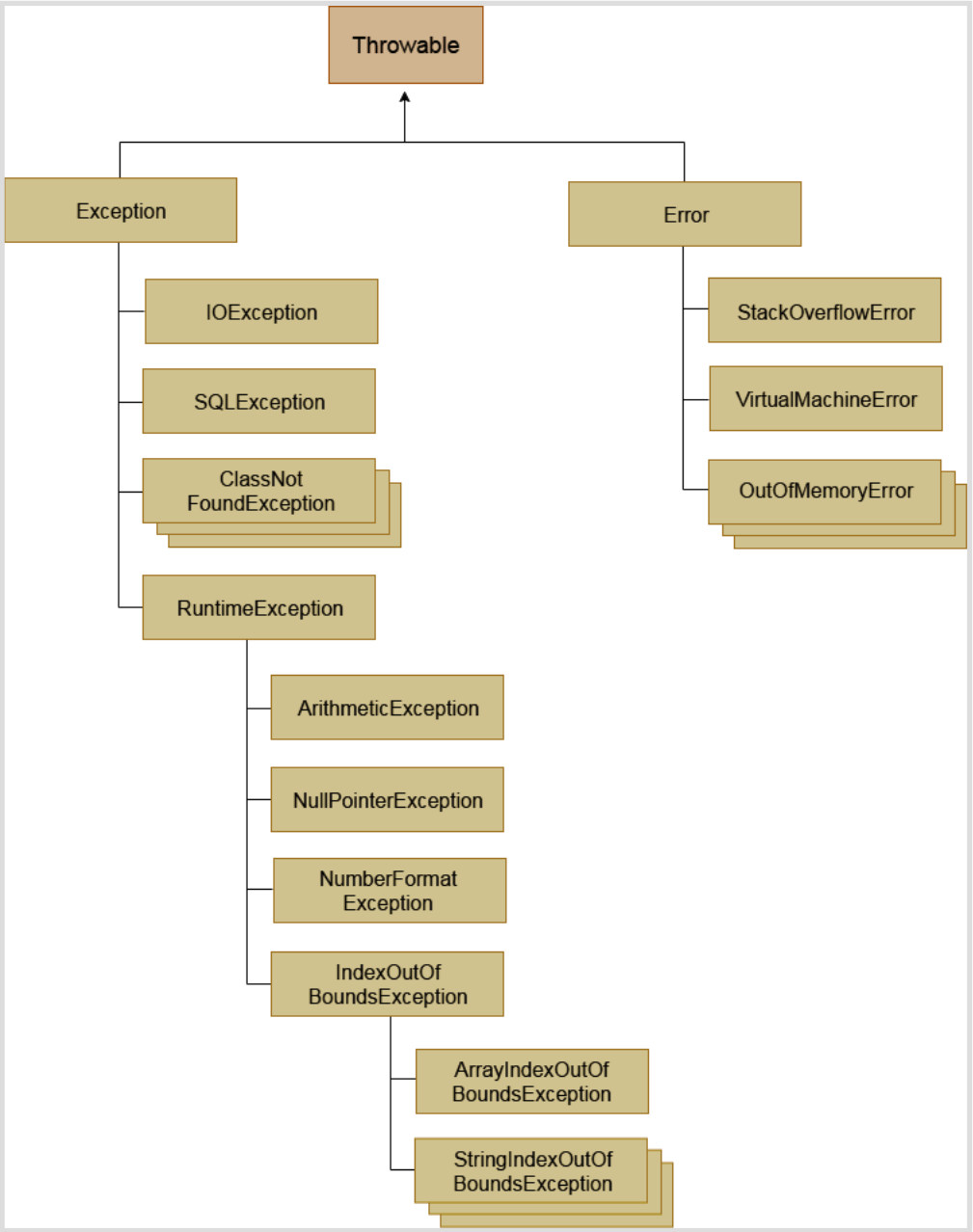
DIFFERENCE BETWEEN ERROR AND EXCEPTION

Error: An Error indicates serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

IN JAVA…

- When an exceptional condition arises
  - An **object** representing that exception is created
  - Then, the **object** is *thrown* in the method that caused the error
  - The method can choose to handle the exception or pass it on
  - In the end, at some point, exception is *caught* and processed
- Who generates the Exceptions?
  - Can be generated by the Java run-time system
  - Can be manually generated by your code


Throwable
Exception
Error
IOException
SQLException
ClassNot FoundException
RuntimeException
StackOverflowError
VirtualMachineError
OutOfMemoryError
ArithmeticException
NullPointerException
NumberFormat Exception
IndexOutOf BoundsException
ArrayIndexOutOf BoundsException
StringIndexOutOf BoundsException

EXCEPTION CLASS

Exception class is used for exceptional conditions that **user programs** should catch

- You will subclass **Exception** to create your own custom exception types
- An important subclass of **Exception** is:
  - **Runtime Exception**: Exceptions of this type are automatically defined for the programs you write
  - Includes: division by zero and invalid array indexing

UNCAUGHT EXCEPTIONS

What happens when you don't handle exceptions in your program ???

```
class uncaught {
public static void main(String args [ ] )
 {
 System.out.println(1/0);
 System.out.println("Hi");
}
```

Exception in thread "main" java.lang.ArithmeticException: /by zero at uncaught.main <uncaught.java:4>

- No compile time error
- Runtime error (Exception)

Example: uncaught.java

UNCAUGHT EXCEPTIONS

When the Java run-time system detects the attempt to divide by zero

- It constructs a new exception object and then *throws* this exception

- What happens next?
  - The execution of **uncaught** stops
  - Exception is *caught* by an exception handler and dealt with immediately

- Where is the exception handler?

UNCAUGHT EXCEPTIONS

Our program should provide the exception handler, catch the exception and process them

- If we haven't supplied any exception handlers of our own
  - The default handler is provided by the Java run-time system
  - Prints a **stack trace** from the point at which the exception occurred
  - **Terminates the program**

EXAMPLE

- The stack trace will always show
  - The sequence of method calls that led up to the error

Example stacktrace:

Exception in thread “main” java.lang.ArithmeticException: / by zero

at div.div1(stacktrace.java:4)at stacktrace.main(stacktrace.java:10)

IN EXAMPLE STACKTRACE

It throws the Exception(ArithmeticException).

Appropriate Exception handler is not found within this method.

JAVA EXCEPTION KEYWORDS

There are five keywords which are used in handling exceptions in Java.

- **try**- The "try" keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone
- **catch**- The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
- **finally**-The "finally" block is used to execute the important code of the program. It is executed whether an exception is handled or not.

- **throw**- The "throw" keyword is used to throw an exception
- **throws**- The "throws" keyword is used to declare exceptions. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

TRY AND CATCH

The default exception handler provided by the Java run-time system is useful for debugging

- Usually, we want to handle an exception ourselves
- **Benefits:**
  - It allows you to fix the error
  - It prevents the program from automatically terminating
- To handle a run-time error
  - Enclose the code that you want to monitor inside a **try** block
  - Immediately following the **try** block, include a **catch** clause
  - Specify the exception type that you wish to catch
  - Example: TCB

TRY AND CATCH CONTD…

In **TCB.java** example:

•**println( )** inside the **try** block is never executed

•Once the **catch** statement has executed

•Program control continues with the next line in the program following the entire **try/catch**

•A **try** and its **catch** statement form a unit

•A **catch** statement cannot catch an exception thrown by another **try** statement

(except in the case of nested **try** statements)

•The statements that are protected by **try** must be surrounded by curly braces

TRY AND CATCH

The goal of most well-constructed **catch** clauses should be

- To resolve the exceptional condition
- Then continue on as if the error had never happened
- Example: ArithmeticException
- Background:

To generate random number: **java.util.Random**

```
Random r = new Random();
int n = r.nextInt();
```

EXCEPTION HANDLING

Lecture 20

Prof. Anita Agrawal

BITS Pilani, Goa

COMMON JAVA EXCEPTIONS

- A scenario where ArithmeticException occurs.

  Example: **int** a=50/0; //ArithmeticException

- A scenario where NullPointerException occurs.

  Example: String s=**null**;

  System.out.println(s.length());

- A scenario where NumberFormatException occurs

  Example: String s="abc";

  int i=Integer.parseInt(s); //NumberFormatException

- A scenario where ArrayIndexOutOfBoundsException occurs-

  Example: int a[ ]=new int [5];

  a [10]=50; //ArrayIndexOutOfBoundsException

TRY-CATCH BLOCK SYNTAX

- try {

  //code that may throw an exception

  } catch(Exception_class_Name ref){ }

- try block is used to enclose the code that might throw an exception. It must be used within the method.
- catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type

MULTIPLE CATCH CLAUSE

- Exception subclasses must come before any of their superclasses
- Otherwise, **catch** statement that uses a superclass will catch exception of that type and its subclasses
- Subclass would never be reached if it came after its superclass
- In Java, unreachable code is an error

- Unreachable code

NESTED TRY

- The try block within a try block is known as 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.
- Each time a **try** statement is entered
- The context of that exception is pushed on the stack
- If an inner **try** statement does not have a **catch** handler for a particular exception
- The next **try** statement's **catch** handlers are inspected for a match
- This continues until one of the **catch** statements succeed
- If no **catch** statement matches
- Then, the Java run-time system will handle the exception

NestC, Nested1

```
//Main try block
try {
  statement 1;
  statement 2;
  //try-catch block inside another try block
  try {
    statement 3;
    statement 4;
    //try-catch block inside nested try block
    try {
      statement 5;
      statement 6;
    }
    catch(Exception e2) {
      //Exception Message
    }
  }
  catch(Exception e1) {
     //Exception Message
  }

}
//Catch of Main(parent) try block
catch(Exception e3) {
    //Exception Message
}
```

FINALLY

- Java finally block is a block that is used to execute important code such as closing connection, stream etc.
- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block
- The **finally** block will execute whether or not an exception is thrown
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception
- Useful for closing file handles and freeing up any other reserved resources
- The **finally** clause is optional
- However, each **try** statement requires at least one **catch** or a **finally**

THROW

- We are only **catching** exceptions that are **thrown by the Java run-time system**
- It is possible for your program to throw an exception explicitly
- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code.
- We can throw either checked or unchecked exception.
- The throw keyword is mainly used to throw custom exceptions.
- The general form of **throw** is

  throw *ThrowableInstance*

THROW

- *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**
- Primitive types and object of **String** and **Object** cannot be used as exceptions.
- The flow of execution stops immediately after the **throw** statement
- The nearest enclosing **try** block is inspected
  - If a **catch** statement that matches the type of exception is available
    - If it does, control is transferred to that statement
    - If not, then the next enclosing **try** statement is inspected, and so on

- If no matching **catch** is found
  - Then the default exception handler halts the program
  - Prints the stack trace

  - Throw1
  - Throw2

THROW

- Most Java's built-in runtime exceptions have at least two constructors
- One with no parameter and
- One that takes a string parameter
- When the second form is used, the argument specifies a string that describes the exception
- This string is displayed when the object is used as an argument to **print( )** or **println( )**

THROWS

- As we know that there are two types of exception checked and unchecked.
- Checked exception (compile time) force you to handle them, if you don't handle them then the program will not compile.
- On the other hand unchecked exception (Runtime) doesn't get checked during compilation.
  - **Throws** keyword is used for handling checked exceptions .
  - By using throws we can declare multiple exceptions in one go.
- The throws does the same thing that try-catch does but there are some cases where you would prefer throws over try-catch.

- Throw4

THROWS

- If a method is capable of causing an exception that it does not handle
- This behavior must be specified to the callers of the method
- A **throws** clause lists the types of exceptions that a method might throw
- General form of a method declaration with a **throws** clause:
  - *type method-name*(*parameter-list*) throws *exception-list*{// body of method}

CREATING YOUR OWN EXCEPTION SUBCLASSES

- Create your own exception types to handle situations specific to your applications
- Just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**)
- Your subclasses don't need to actually implement anything
- Their existence in the program allows you to use them as exceptions
- The **Exception** class does not define any methods of its own
- It inherits the methods provided by **Throwable**
- You may also wish to override one or more of these methods

- **Exception** defines multiple public constructors
- Two most commonly used constructors
  - *Exception( )* -creates an exception that has no description
  - *Exception(String msg)* -lets you specify a description of the exception

Threads and Multithreading

Prof. Anita Agrawal
BITS Pilani k.K.Birla Goa campus

Multithreading

- Multithreading is a feature that allows concurrent execution of two or more threads of a process for maximum utilization of CPU.

- So, threads are light-weight processes within a process.

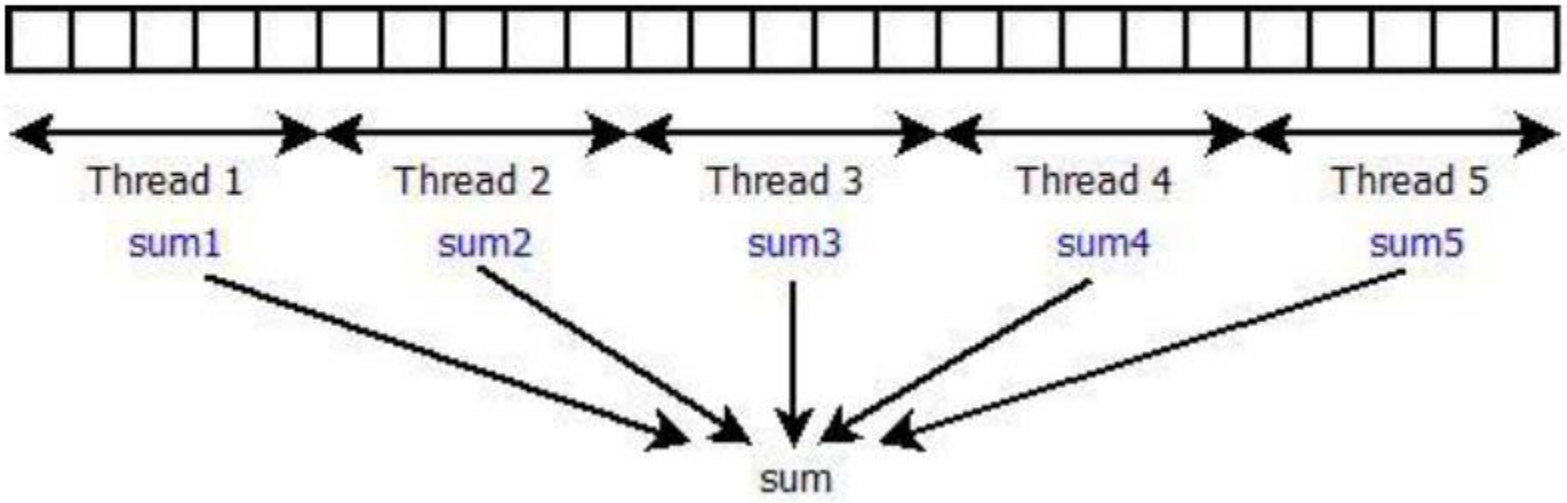
- A thread shares with other threads belonging to the same process the
  - Code section
  - Data section
  - Operating-system resources, such as open files

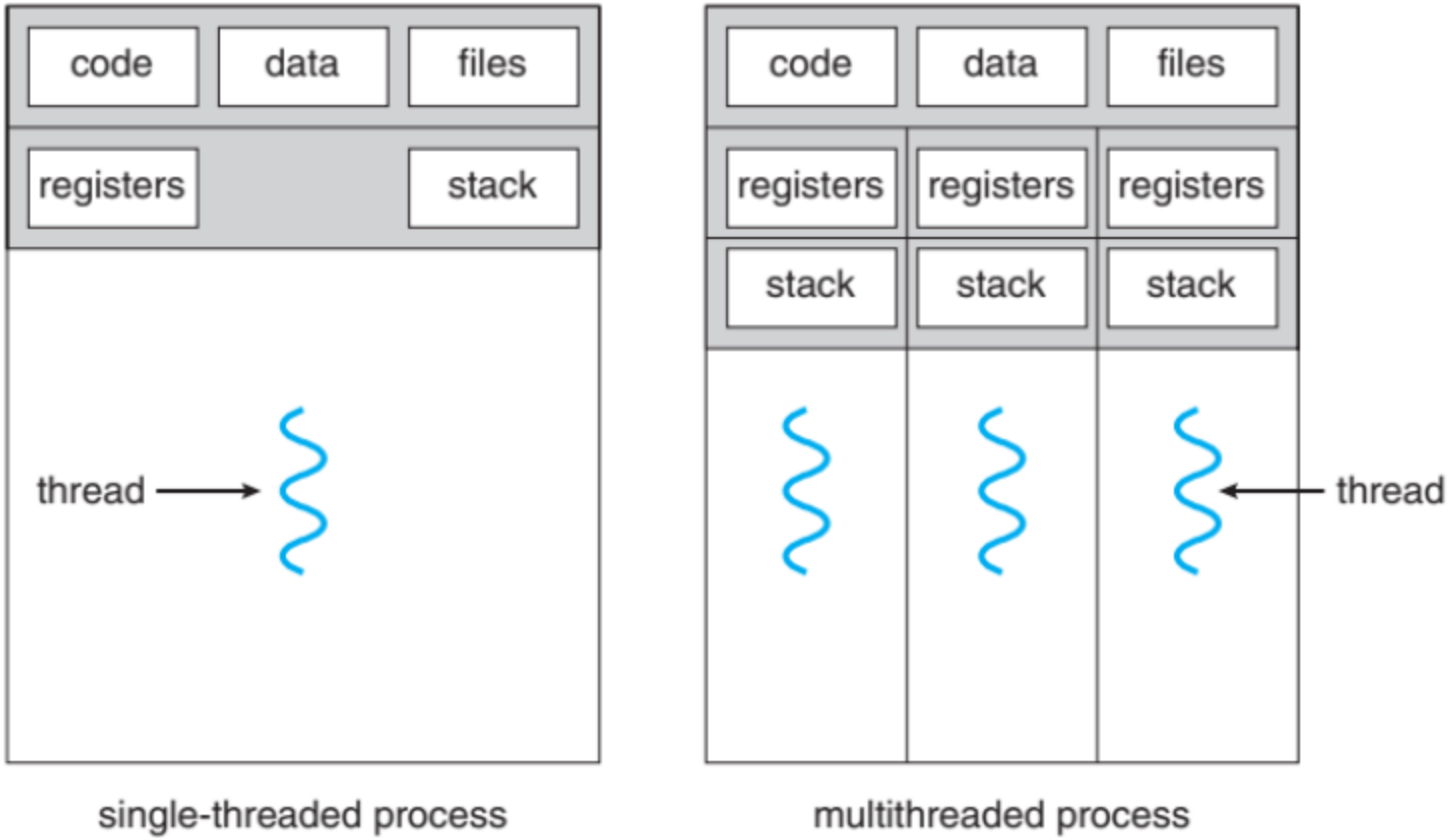
Example 1: A media player

- A media player, where
  - one thread is used for opening the media player,
  - one thread for playing a particular song and
  - one thread for adding new songs to the playlist.

Example 2: Array sum

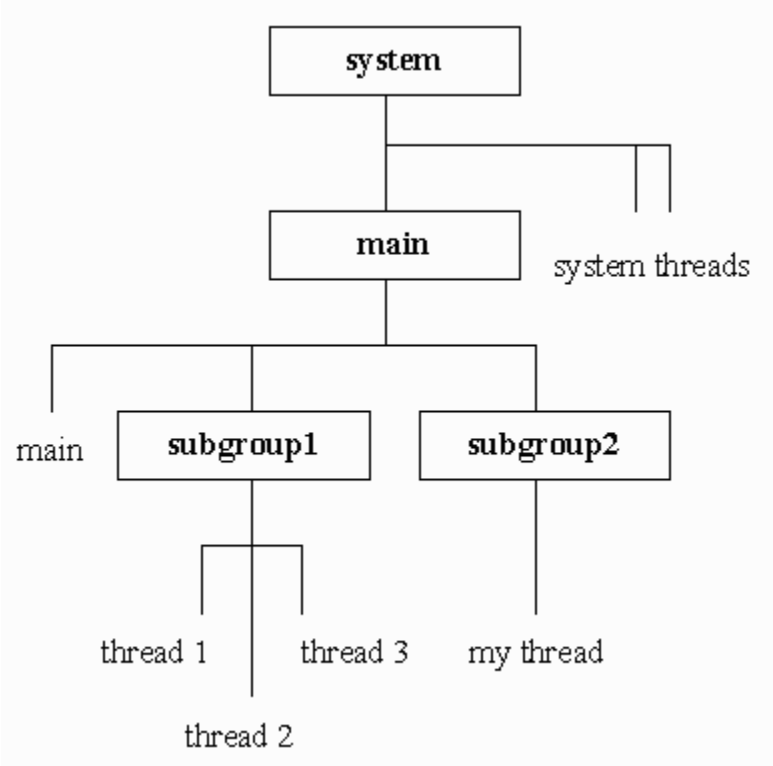
We can think of threads as child processes that share the parent process resources but execute independently.




code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

The Main Thread

- When a Java program starts up, one thread begins running immediately
- This is usually called the *mainthread* of your program
- The main thread is important for two reasons
  - It is the thread from which other “child” threads will be spawned
  - Often, it must be the last thread to finish execution because it performs various shutdown actions
- The main thread is created automatically when your program is started



An application's hierarchical thread-group structure begins with a main thread group just below the system thread group

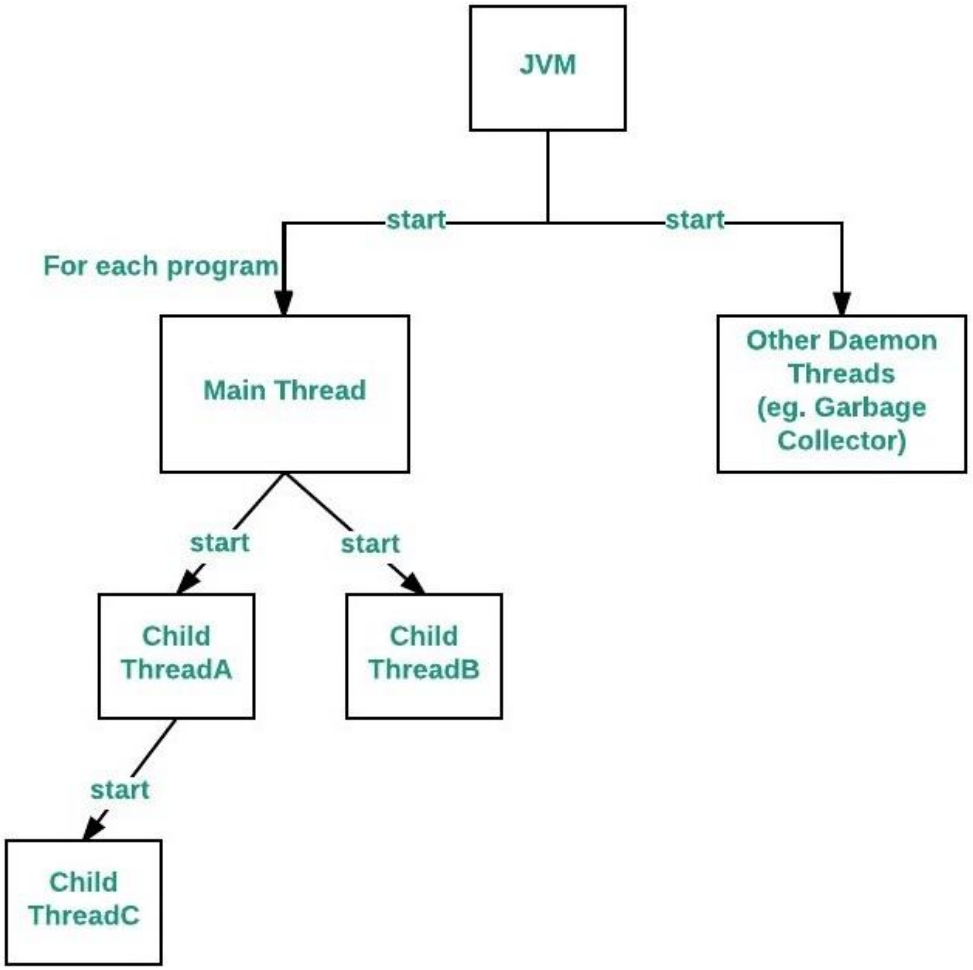
Thread Group

- Java requires every thread and every thread group—save the root thread group, system—to join some other thread group

- **System thread group:** The JVM-created system group organizes JVM threads that deal with object finalization and other system tasks, and serves as the root thread group.

- **Main Thread group**: JVM-created main thread group, which is system's subthread group (subgroup, for short).
  - main contains at least one thread that executes byte-code instructions in the main() method.
- Subgroup 1 and Subgroup 2 subgroups: Application-created subgroups

- Thread 1, Thread 2, and Thread 3: Subgroup 1's three application-created threads
- my thread: subgroup 2 group's one application-created thread


JVM
start
start
For each program
Main Thread
Other Daemon Threads (eg. Garbage Collector)
start
start
Child ThreadA
Child ThreadB
start
Child ThreadC

- You can control the main thread through a **Thread** object

- To do so, you must obtain a reference to it by calling the method
  currentThread ( )
  which is public static member of **Thread** class

- This method returns a reference to the thread in which it is called.

- Once you have a reference to the main thread, you can control it just like any other thread.

Main Thread Example

- A reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable **t**.
- Next, the program displays information about the thread.
- The program calls **setName( )** to change the internal name of the thread
- Information about the thread is then redisplayed
- Next, the loop downcounts from 5 with a pause of 1s in between two consecutive counts
- The thread execution can be paused by the **sleep( )** method

- The argument to **sleep( )** specifies the delay period in milliseconds
- We use **try/catch** block around the loop
- The **sleep( )** method in **Thread** might throw an **Interrupted Exception**
- This would happen if some other thread wanted to interrupt this sleeping one

Main Thread Example:Output

- The output produced when Thread object **t** is used as an argument to **println( )**
- This displays, in order: the **name of the thread, its priority, and the name of its group**
- By default, the name of the main thread is **main**
- Its priority is 5, which is the default value (Highest: 10, Lowest: 1)
- **main** is also the name of the group of threads to which this thread belongs
- You can obtain the name of a thread by calling **getName( )**

Threading in Java

- Threads can be created by two mechanisms:
  - **By implementing the Runnable interface**
  - **By Extending the Thread class**

- To create a new thread, your program will either **extend Thread** or **implement Runnable** interface

- All the examples that have been used a single thread of execution

Creating a Thread: Implementing Runnable interface

- A class need only implement a single method called **run()**

```
public void run()
```

- Inside **run( )**, you will define the code that constitutes the new thread
- **run( )** can call other methods, use other classes, and declare variables, just like the main thread can
- The only difference is that **run( )** establishes the entry point for another thread of execution within your program

Implementing Runnable Interface

- Create a class that implements **Runnable**
- Instantiate an object of type **Thread**
- **Thread** defines several constructors.
- After the new thread is created, it will not start running until you call its **start( )** method

start( ) executes a call to **run( )**

Thread1.java, Thread7.java

- 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().
- **public void run():** is used to perform action for 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.

Thread: Constructor Overloading

- Thread():
- Thread(Runnable target):
- Thread(Runnable target, String name):
- Thread(String name):
- Thread(ThreadGroup group, Runnable target):
- Thread(ThreadGroup group, Runnable target, String name):
- Thread(ThreadGroup group, Runnable target, String name, long stackSize):
- Thread(ThreadGroup group, String name):

Extending Thread class

The second way to create a thread is

- Create a new class that extends **Thread**
- Then create an instance of that class
- The extending class must override the **run( )** method
- It must also call start() to begin execution of the program

Stages of lifecycle of threads

- A thread goes through various stages in its life cycle.
  - For example, a thread is born, started, runs, and then dies.

Stages of Lifecycle of Threads

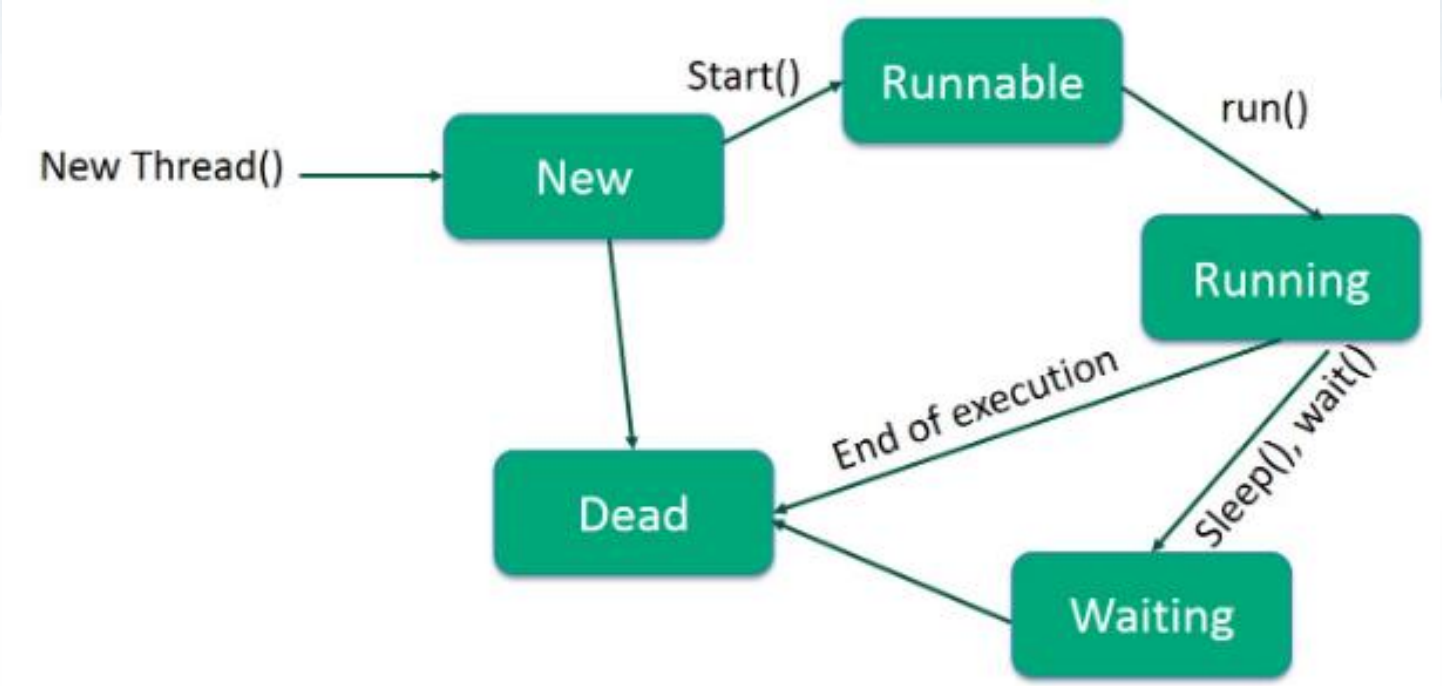
- **New** - When we create an instance of Thread class, a thread is in a new state. A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.

- **Running -** The Java thread is in running state. A thread in this state is considered to be executing its task.

- **Waiting-** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

- **Suspended** - A running thread can be **suspended**, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off.
- **Blocked** - A Java thread can be blocked when waiting for a resource.
- **Terminated** - A runnable thread enters the terminated state when it completes its task or otherwise terminated.
  - A thread can be terminated, which halts its execution immediately at any given time. Once a thread is terminated, it cannot be resumed.


New Thread()
New
Start()
Runnable
run()
Running
End of execution
Sleep(), wait()
Dead
Waiting

Choosing an approach

- Which one is better….Implementing Runnable or extending class????
- If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
- We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

Using isAlive() and join()

- Often you will want the main thread to finish last
  - In the preceding examples, this is accomplished by calling **sleep( )** within **main( )**

- How can one thread know when another thread has ended?
  - **Thread** provides a means by which you can answer this question

    isAlive( ) method defined by **Thread**

    final boolean isAlive( )

- **isAlive( )** is occasionally useful

Using isAlive and Join

- Method that you will more commonly use to wait for a thread to finish is called **join( )**
  - final void join( ) throws InterruptedException
- **join( )**method waits until the thread on which it is called terminates
- It's name comes from the concept of the calling thread waiting until the specified thread *joins* it
- **join( )** allows you to specify a maximum amount of time that you want to wait for the specified thread to terminate

- public void run(): is used to perform action for a thread.
- public void start(): starts the execution of the thread. JVM calls the run() method on the thread.
- public void sleep(long miliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- public void join(): waits for a thread to die.
- public void join(long miliseconds): waits for a thread to die for the specified miliseconds.
- public int getPriority(): returns the priority of the thread.
- public int setPriority(int priority): changes the priority of the thread.

- public String getName(): returns the name of the thread.
- public void setName(String name): changes the name of the thread.
- public Thread currentThread(): returns the reference of currently executing thread.
- public int getId(): returns the id of the thread.
- public Thread.State getState(): returns the state of the thread.

- public boolean isAlive(): tests if the thread is alive.
- public void yield(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
- public void suspend(): is used to suspend the thread(deprecated).
- public void resume(): is used to resume the suspended thread(deprecated).

- public void stop(): is used to stop the thread(depricated).
- public boolean isDaemon(): tests if the thread is a daemon thread.
- public void setDaemon(boolean b): marks the thread as daemon or user thread.
- public void interrupt(): interrupts the thread.
- public boolean isInterrupted(): tests if the thread has been interrupted.
- public static boolean interrupted(): tests if the current thread has been interrupted.

Java IO and file Handling

Prof. Anita Agrawal

BITS Pilani- K.K.Birla Goa Campus

Email: aagrawal@goa.bits-pilani.ac.in

fppt.com

Introduction

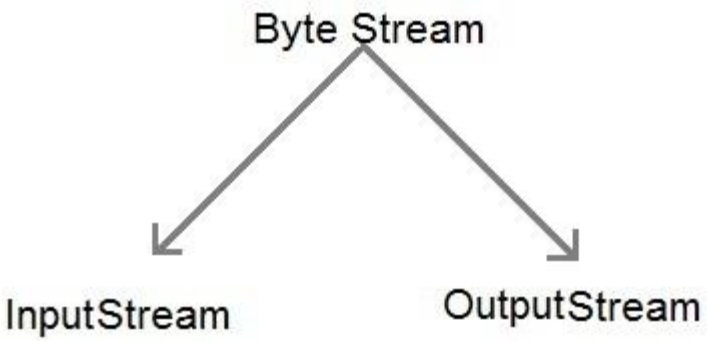
- **Java I/O** (Input and Output) is used *to process the input* and *produce the output*.
- Java uses the concept of a **stream** to make I/O operations fast.
- A stream is linked to a physical layer by java I/O system to make input and output operations.
- Java encapsulates Stream under **java.io** package
- The java.io package contains all the classes required for input and output operations.
- File handling in Java is performed by Java I/O

fppt.com

Types of Streams

- Two types of streams:
  - Byte Stream
  - Character Stream
- Byte stream:

  Defined by using two abstract classes at the top of hierarchy, they are InputStream and OutputStream.



fppt.com

Byte Oriented streams and character-oriented streams

- Byte-oriented streams.
  - Intended for general-purpose input and output.
  - Data may be primitive data types or raw bytes.
- Character-oriented streams.
  - Intended for character data.
  - Data is transformed from/to 16 bit Java char used inside programs to the UTF format used externally.

fppt.com

Byte Stream

- Create binary files.
- A binary file essentially contains the memory image of the data. That is, it stores bits as they are in memory.
- Binary files are faster to read and write because no translation need take place.

fppt.com

InputStream

- Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.
- It is an abstract class.
- It is the superclass of all classes representing an input stream of bytes.

fppt.com

Useful methods of input stream

Method	Description
1) public abstract int read()	reads the next byte of data from the input stream. It returns -1 at the end of the file.
2) public int available()	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()	is used to close the current input stream.

fppt.com

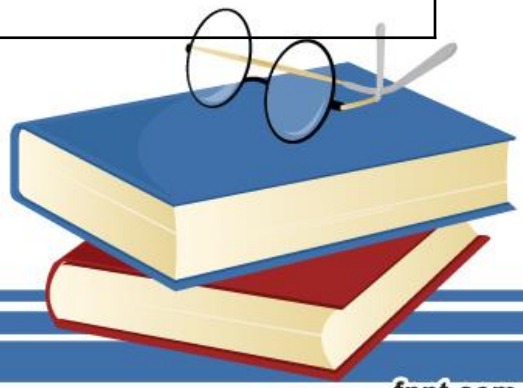
OutputStream class

- Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.
- It 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.

fppt.com

Important methods of output stream

Method	Description
1) public void write(int)	is used to write a byte to the current output stream.
2) public void write(int[ ])	is used to write an array of byte to the current output stream.
3) public void flush()	flushes the current output stream.
4) public void close()	is used to close the current output stream.



fppt.com

Important Byte Stream classes

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain print() and println() method

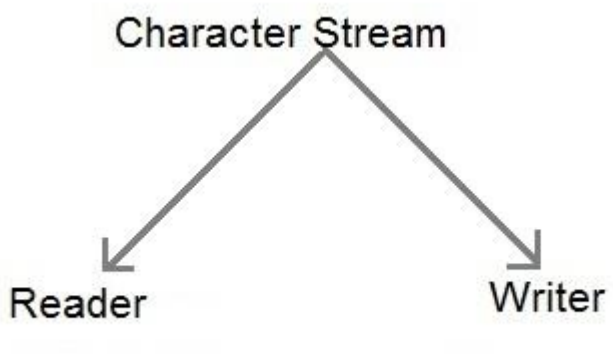
fppt.com

- Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream.**

fppt.com

Character Stream

- Character stream is also defined by using two abstract classes at the top of hierarchy, they are Reader and Writer.



fppt.com

Character Streams

- **Character** streams are used to perform input and output for 16-bit unicode
- Character streams create text files.
- These are files designed to be read with a text editor.
- Java automatically converts its internal unicode characters to the local machine representation (ASCII in our case).
- These two abstract classes have several concrete classes that handle unicode character.

fppt.com

Some important character stream classes

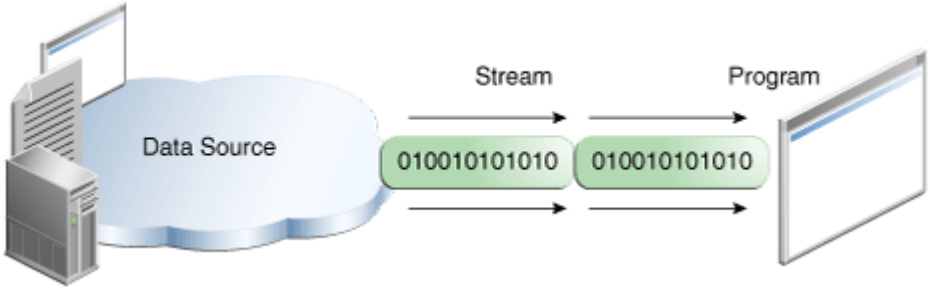
Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain print() and println() method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

fppt.com

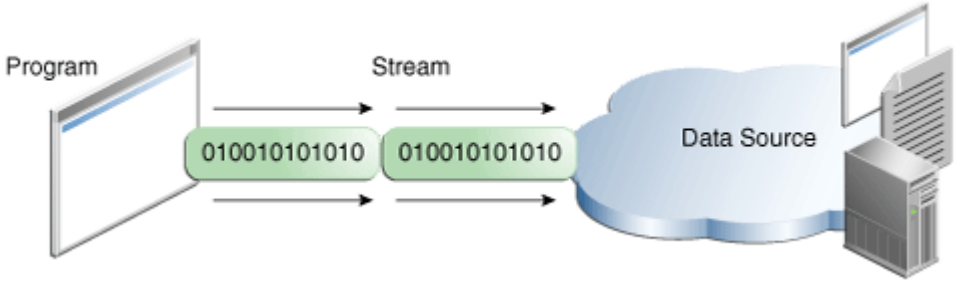
- Though there are many classes related to character streams but the most frequently used classes are, **Reader** and **Writer.**

fppt.com

Reading information into a program



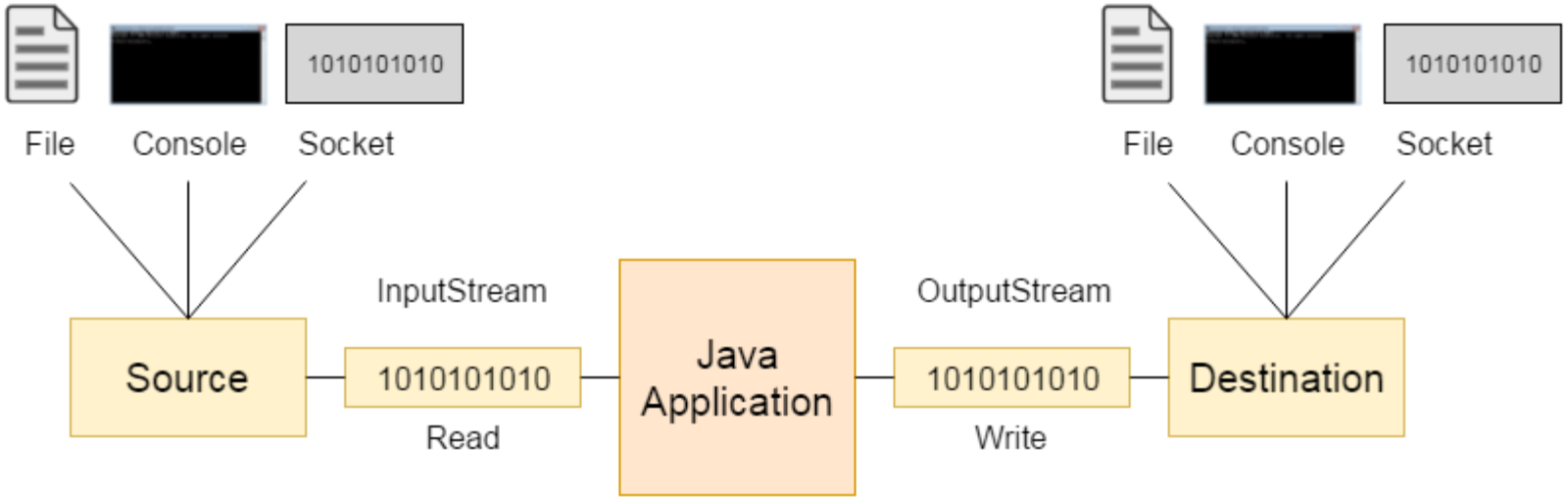
Writing information from a program



No matter how they work internally, all streams present the same simple model to programs that use them

The data source and data destination pictured above can be anything that holds, generates, or consumes data. This includes disk files, another program, a peripheral device, a network socket, or an array.

Working of Java input and output stream



fppt.com

Reading and Writing a file

- To open a file, you simply create an object of one of these classes, viz:
- FileInputStream
- FileOutputStream
- Specifying the file name as an argument to the constructor
  - FileInputStream(String fileName)
  - FileOutputStream(String fileName)

fppt.com

- For input stream:
  - If the file **does not exist,** then **FileNotFoundException** is thrown
- For output stream:
  - If the file cannot be **opened** or **created,** then **FileNotFoundException** is thrown
- **FileNotFoundException** is a subclass of **IOException**
  - So catching an **IOException** exception will do the job
- When an output file is opened, any pre-existing file by the same name is destroyed

fppt.com

Reading from a file

- To read from a file
  - Use **read( )** method defined within **FileInputStream**
    - Int read( )
- Each time **read()** is called
  - A single byte is read from the file and returns the byte as an integer value
  - **read( )** returns –1 when the end of the file is encountered
  - **read( )**can throw an **IOException**

Read1.java, Read2.java

fppt.com

- **NOTE:**
  - Resource declared in the **try** statement is implicitly **final**
  - You can't assign to the resource after it has been created
  - We can manage more than one resource within a single **try** statement with a semicolon

Read8.java

fppt.com

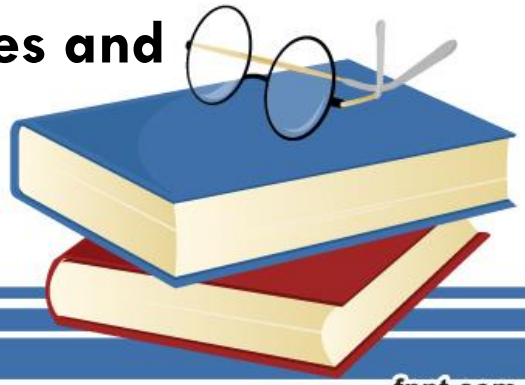
Closing a file

- When you are done with a file, you must close the file
  - Closing a file releases the system resources allocated to the file
  - Allowing them to be used by another file
- The traditional approach to close the file is by calling the **close( )** method
  - void close( )
  - The second is to use the **try**-with-resources statement
  - Automatically closes a file when it is no longer needed
  - No explicit call to **close( )** is executed
  - Available for JDK 7 or later

fppt.com

Automatically closing a file revisited…..

- Explicit calls to **close( )** are made once a file is no longer needed
- JDK 7 added a new feature that offers another way
  - This feature, sometimes referred to as *automatic resource management*(**ARM**)
  - Simply, an expanded version of the **try** statement
- Try-with-resources
  - try (resource-specification) {// use the resource}
- *resource-specification* is a statement that **declares and initializes** a resource, such as a file stream



fppt.com

Automatically closing a file

```
try(FileInputStream fr = new FileInputStream(args[0])) {
```

- **try** declares a **FileInputStream** called **fr**
- **fr** is then assigned a reference to the file opened by its constructor
- The variable **fr** is local to the **try** block being created when the **try** is entered
- When the **try** is left, the stream associated with **fr** is **automatically** closed

fppt.com