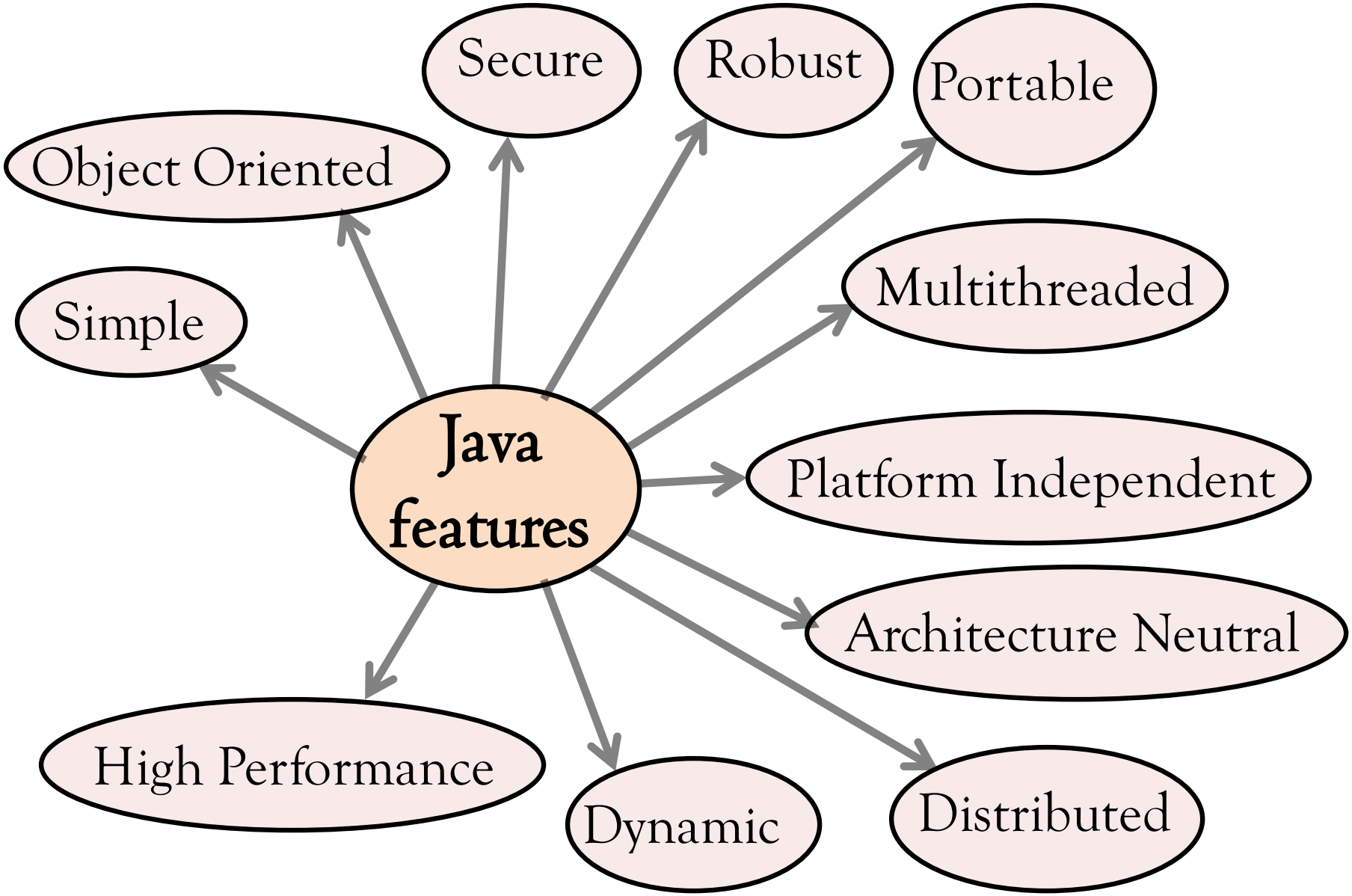
Java

- ver 1.8

Java features



contd..

- **Simple** – very easy program structure and few key words.
- **Object Oriented** – Java practices all Object programming concepts to the core viz.. abstraction, encapsulation, inheritance, polymorphism.
- **Platform independence** – java program execute from JVM and not connected directly to any OS.
- **Architecture Neutral** – The system features of java e.g. memory requirements remain same irrespective of CPU being 16 – bit or 32 bit.
- **Secure** – Java offers security at levels of compilation, execution. Provides an API called java.security.
- **Robust** – Java does strict type checking, offers exception handling to prevent application crash.

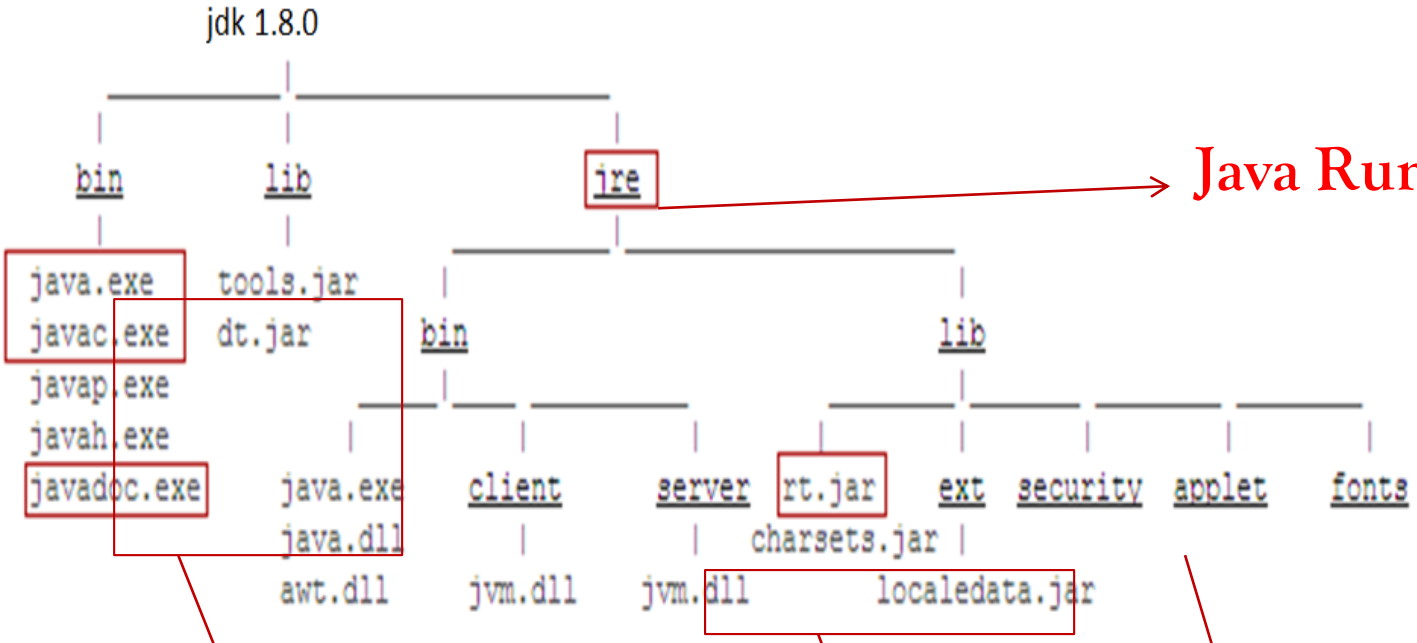
contd..

- **Portability** – The feature using which user can transport the java code to any machine and execute without re–compilation. Also aids reusability.
- **Distributed** – Java provides APIs to develop distributed applications using client–server, multi–tiered architecture.
- **Dynamic** – Java provides ready made utilities for dynamic memory allocation for objects.
- **High Performance** – The bytecode has been carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. JRE systems that provide this feature lose none of the benefits of the platform-independent code.

Terminologies

- **JDK – Java Development Kit**: contains files and tools that can be used to develop and execute programs, applications, etc;
- **JRE – Java Runtime Environment**: contains files and tools that are used to execute programs, applications viz.. javac, java, appletviewer. Consists of JVM, Java API.
- **JVM – Java Virtual Machine**: contains files and tools that are used to execute class files. It contains class loader.
  - **Byte code verifier** – verifies the byte code for any errors
  - **Class loader** – a program that loads the class file into the JVM
  - **JIT Compiler** – Just-In-Time compiler, compiles the class file.
- **JAR – Java Archive :** A file format that is used to package class files for deployment. The class files are packaged into archives.

JDK installation directory



Java Runtime Environment

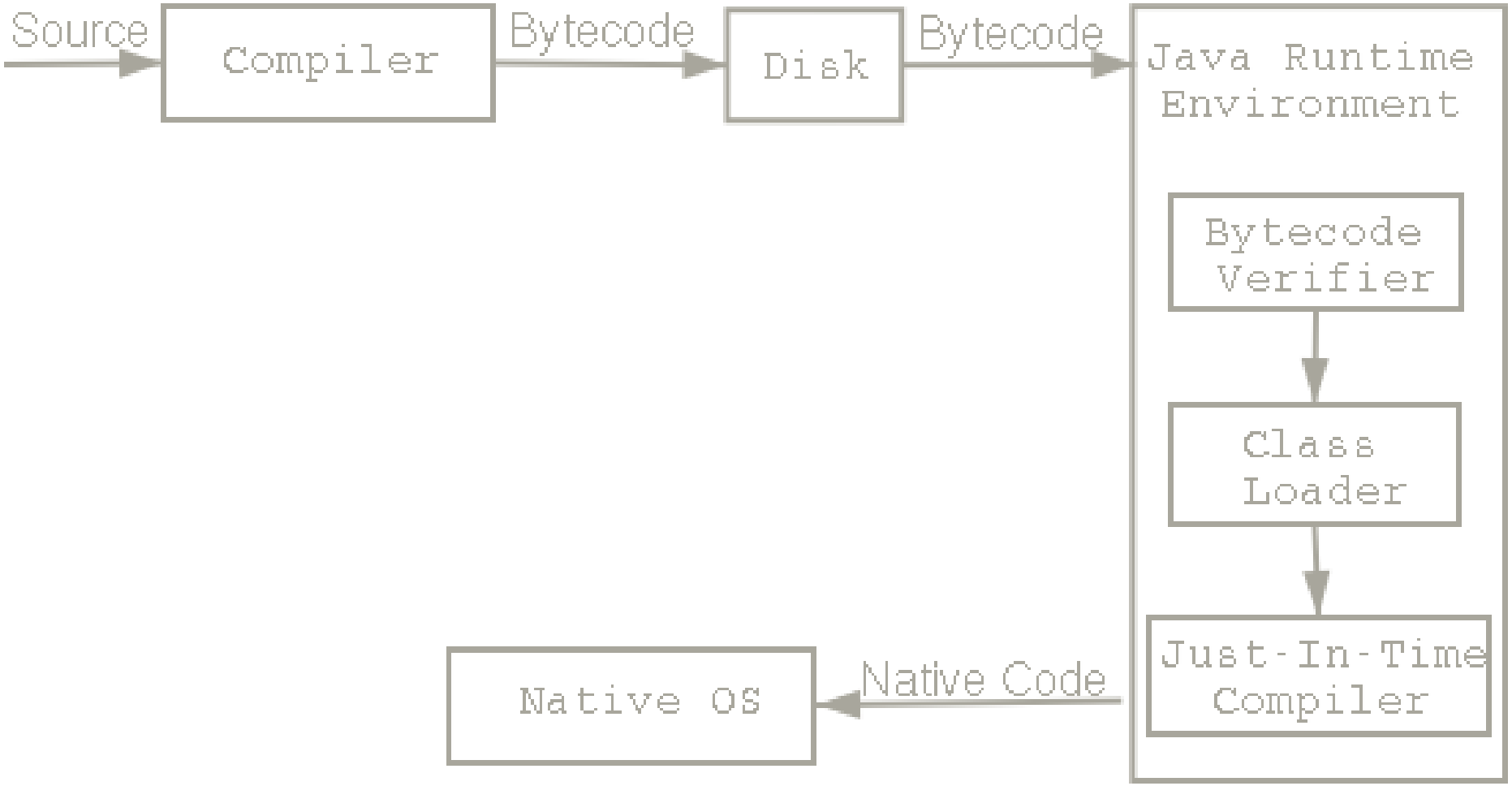
Java Executables

Java virtual machine

Java Predefined Classes like String, Date, Math etc.

contd..

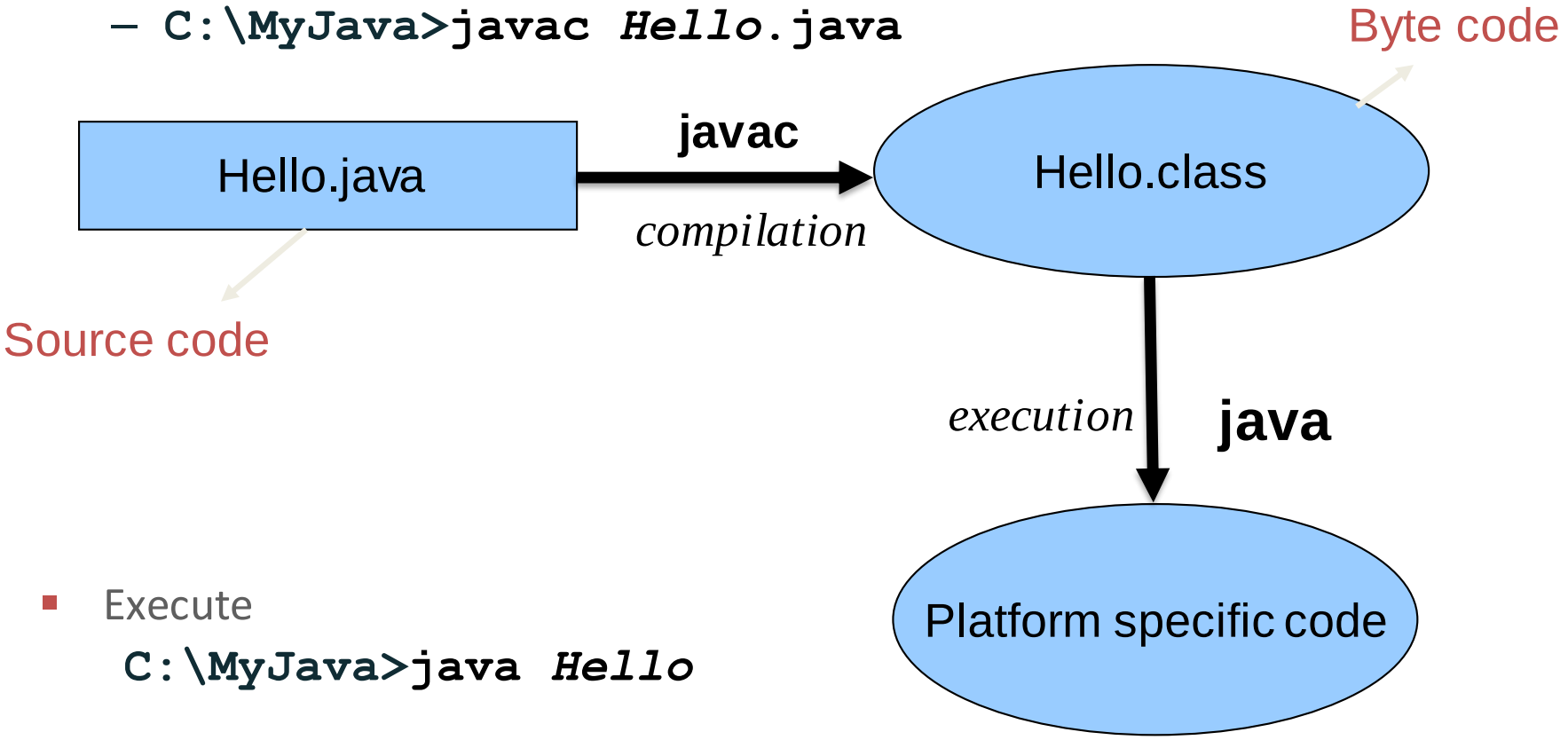
- General program structure.
- Java is compiled and interpreted. This is possible because of the concept of **ByteCode**.



Compilation & execution : on console

Save the file as Hello.java. A public class must be saved in the same name as class name.

- Compile
  - `C:\MyJava>javac Hello.java`



- Execute

  `C:\MyJava>java Hello`

JRE and Bytecode

- Java Bytecode is produced when Java programs are compiled.
- To execute Java program, JRE must be installed in the system
- JRE or Java Runtime environment contains
  - ✓ Java Virtual Machine
  - ✓ Standard class libraries (APIs)
  - ✓ Java Plug-in
  - ✓ Java Webstart
- JRE gets installed automatically when JDK is installed. JRE can be installed independent of JDK as well. This will mean that Java programs can be executed in that system.

contd..

- Java is case sensitive.
- Variables
  - The character set called as **Unicode**.
  - Naming
- **Scope & Life time** – program elements have following scopes.
  - block
  - method
  - class and subclass
  - package
- **Note** : Though blocks can be nested, its not possible to declare a variable to have the same name as one in an outer scope.
- Type conversion and casting
- Comments – single line and multi line comments.

Java Environment

- Java Environment setup involves setting up of environment variables viz.. classpath.
- CLASSPATH is an environment variable instructs the,
  - Java compiler javac.exe to locate class files to be imported.
  - JVM class loader to find the classes that are directly or indirectly invoked, including the system classes.
- Setting the CLASSPATH variable
  - **Windows :** SET CLASSPATH= to set the CLASSPATH variable. E.g. SET CLASSPATH= C:\jdk1.5.0_06\bin
  - **Linux :** Modify the **.bash_profile (hidden)** file as follows:
    - ✓ export CLASSPATH=<classpath1>:<installation_path>/bin\

Java versions

Version	Features
1.0	Initial release
1.1	Inner classes; RMI, JDBC;
1.2	New keyword – strictfp; Swing graphical API integrated into the core classes; Sun's JVM was equipped with JIT Compiler.
1.3	RMI was modified to support optional compatibility with CORBA; JNDI included in core libraries.
1.4	New keyword – assert; 64 bit support; Due to exception chaining, exceptions can encapsulate original lower–level exception; Internet Protocol ver – 6 support non-blocking; Image I/O API for reading and writing images; Integrated XML parser and XSLT processor; Integrated security and cryptography extensions ; Java Web Start included

contd..

1.5	Generics; Autoboxing; Annotations; Collections; Enumerations; auto stub generation for RMI;
1.6	Enhanced support for Web services.
1.7	Enhancements – underscore in integer variables, String values in switch statement, Automatic resource management i.e. can be mentioned with try block.
1.8	Lambda expressions, default methods in interfaces, method referencing, Optional class, Collectors class, ForEach() method, Stream API.

Data types and Variables

Category	Data type	Memory Size	Default Value
Numeric	byte	1	0
	short	2	0
	int	4	0
	long	8	0L
	float	4	0.0f
	double	8	0.0d
Character	char	2	'\u000'
	String		null
Boolean	boolean		false

contd..

- Java supports a few special escape sequences for char and String literals:
  - \b backspace,
  - \t tab,
  - \n line feed,
  - \f form feed,
  - \r carriage return,
  - \" double quote,
  - \‘ single quote, and
  - \\ backslash.

Operators

Sl.No	Category	Operation	Operators
1.	Unary	Increment	++
		Decrement	--
2.	Binary	Relational	>, <, >=, <=, ==, !=
		Logical	&&, ||, !
	Note : Java supports Short Circuit logical operations.		
		Arithmetic	%, /,*, +,-
		Assignment	=
3.	Bitwise	Bitwise Logical	&, |, ~ ,^, &=, |=,^=,
		Bitwise Shift	>>, <<, >>>, >>=,<<=,>>>=

contd..

Sl.No	Category	Operation	Operators
4.	String	Concatenation	+
5.	Memory	Allocation	new
6.	Compound	Arithmetic	%=, /=, *=, +=, -=
7.	Ternary	Conditional	?:

Control Statements

Sl.No	Category	Operation
1.	Conditional	❖ if ❖ if…else ❖ else..if ladder ❖ switch
2.	Repetitive	❖ while loop ❖ do..while ❖ for loop
3.	Enhanced for loop	❖ for(type itr var : collection) { statements ; }
Key words in control statements		❖ continue ❖ break ❖ default

- **Nesting** – Nesting is permitted in all the statements.

Arrays

- Arrays are objects.
- Arrays are of two types – single dim and multi dim.
- Arrays are stored contiguously.
- Java provides for strict upper limit checking.
- Arrays – Array declaration is a two step process.
- Array processing is a two step process.
  - Step1 : Declaration

    int a[];
  - Step2 : Initialization

    a = new int[5];

contd..

- **Multidimensional Arrays**
  - Dynamic nature of multi–dim arrays.
  - As with declaration of single dimension array, multi–dimension array declaration is also a two step process.
  - Array processing is a two step process.
    - ✓ Step1 : Declaration

      int a[][];
    - ✓ Step2 : Initialization

      a = new int[2][3];
- Arrays can be initialized in the following ways :
  - int a[] = { 10,20,30,40,50};
  - int [] a = { 10,20,30,40,50};
  - int a[][2]
  - int 2[]

Classes
&
Objects

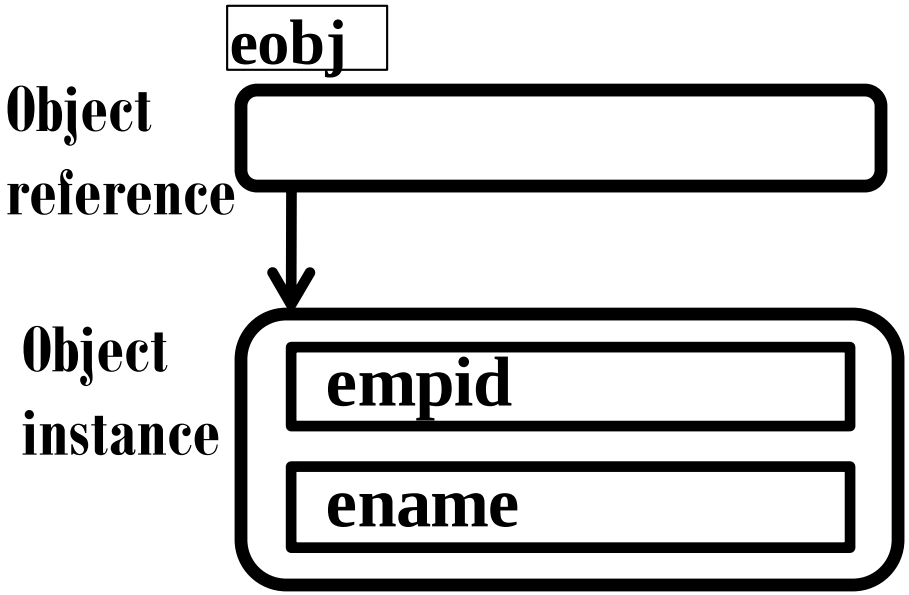
Classes and Objects

Class	Object
➢ **template** for objects definition ➢ contains data members and method definitions ➢ each is specified with **access specifier** and **type qualifier** ➢ defines a system and its properties viz.. polymorphism, abstraction, reusability and like.	➢ **instance** of a class ➢ defines **independent memory location** ➢ has access to all properties of the class according to access rules. ➢ is initialized by special method called constructor and has state

contd..

Diagrammatic representation of object definition

```
class Employee
{
 int empid;
 String ename;
 void input()
 { empid=101; ename="Akash"; }
 void output()
{ System.out.println(empid);
 System.out.println(ename); }
}
public static void main(String[] a)
{  Employee eobj;
   eobj = new Employee();
  }
```



contd..

- **Object execution sequence –**
  1. Declaration
  2. Execution
  3. Garbage collected.
- **Object life cycle –**
  1. Declaration
  2. Initialization
  3. Input
  4. Process
  5. Output
  6. Garbage collected

contd..

Accessibility – Java provides for four access specifiers viz..

Sl.No	Accessibility	Applied to
1.	public	instance variables, methods, class
2.	protected	instance variables, methods, class
3.	default	instance variables, methods, class
4.	private	instance variables, methods

contd..

- **Instance variables**– they are the data members of the class
- **Methods**– are of two categories viz.. accessor and mutator
  - Accessor – thru which the state of the object is accessed.
  - Mutator – thru which the state of the object is modified.
  - A method (or function) in Java has these main parts:
    1. Name
    2. Parameter list
    3. Body
    4. return type.
- **Constructors**– are special methods which initialize objects
  - have same name as that of the class;
  - can be default or explicit;
  - can be overloaded;
  - can be final, synchronized

contd..

- **this –** a reference variable which is member of the class. It is implicitly declared by the JVM and is of the class type.
  - this always points to the current object which is active inside the class.
- **Garbage collection**– done via a method finalize().

contd..

Sl.No	Qualifier	Applied to
1.	**static**	instance variables, methods, blocks, objects, class
2.	**final**	instance variables, methods, objects, class
3.	**transient**	instance variables, objects
4.	**volatile**	instance variables, objects.
5.	**synchronized**	methods, blocks, objects.
6.	**abstract**	methods, class.

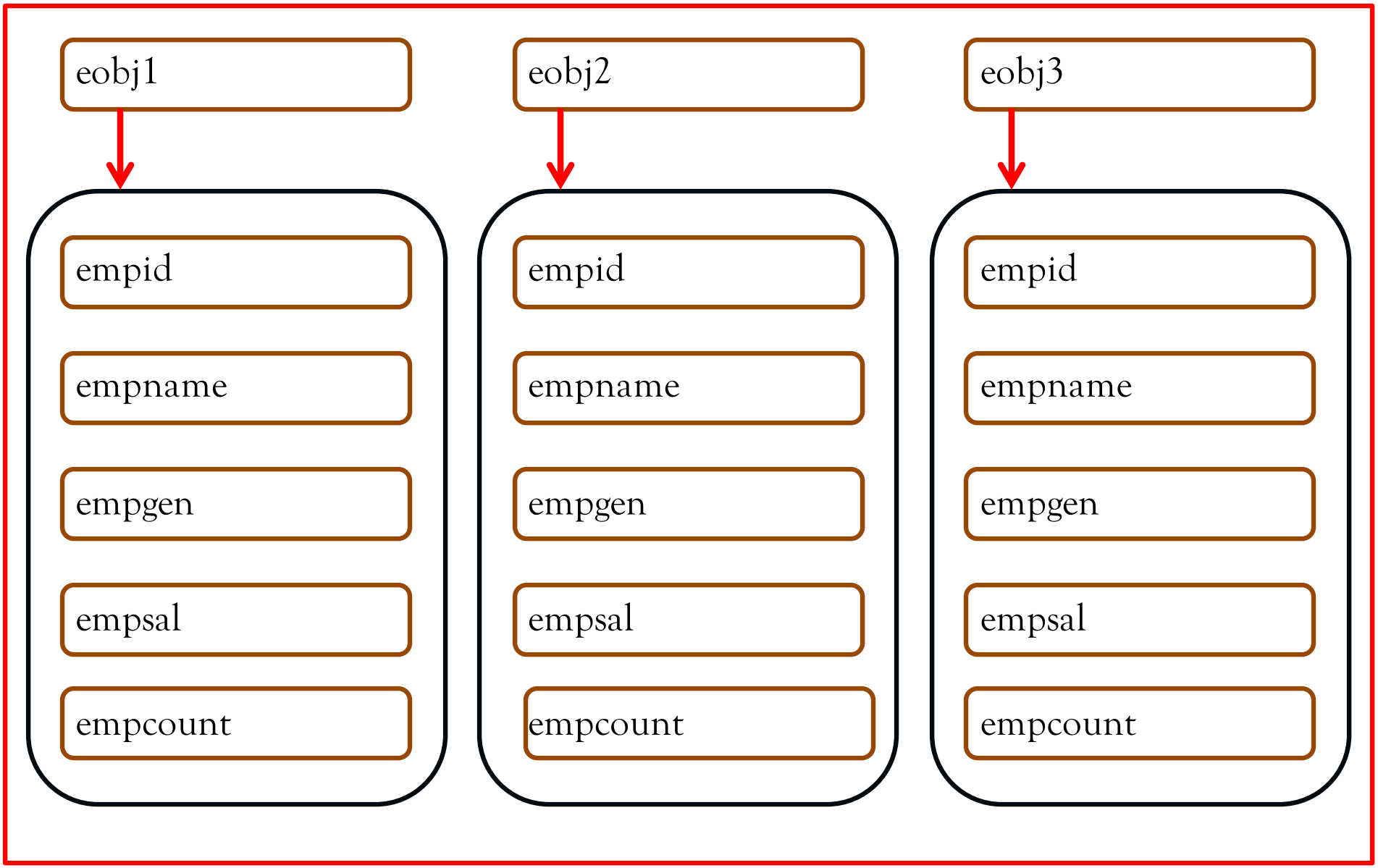
Note – a data member can be static and final; but final assumes greater priority.

static

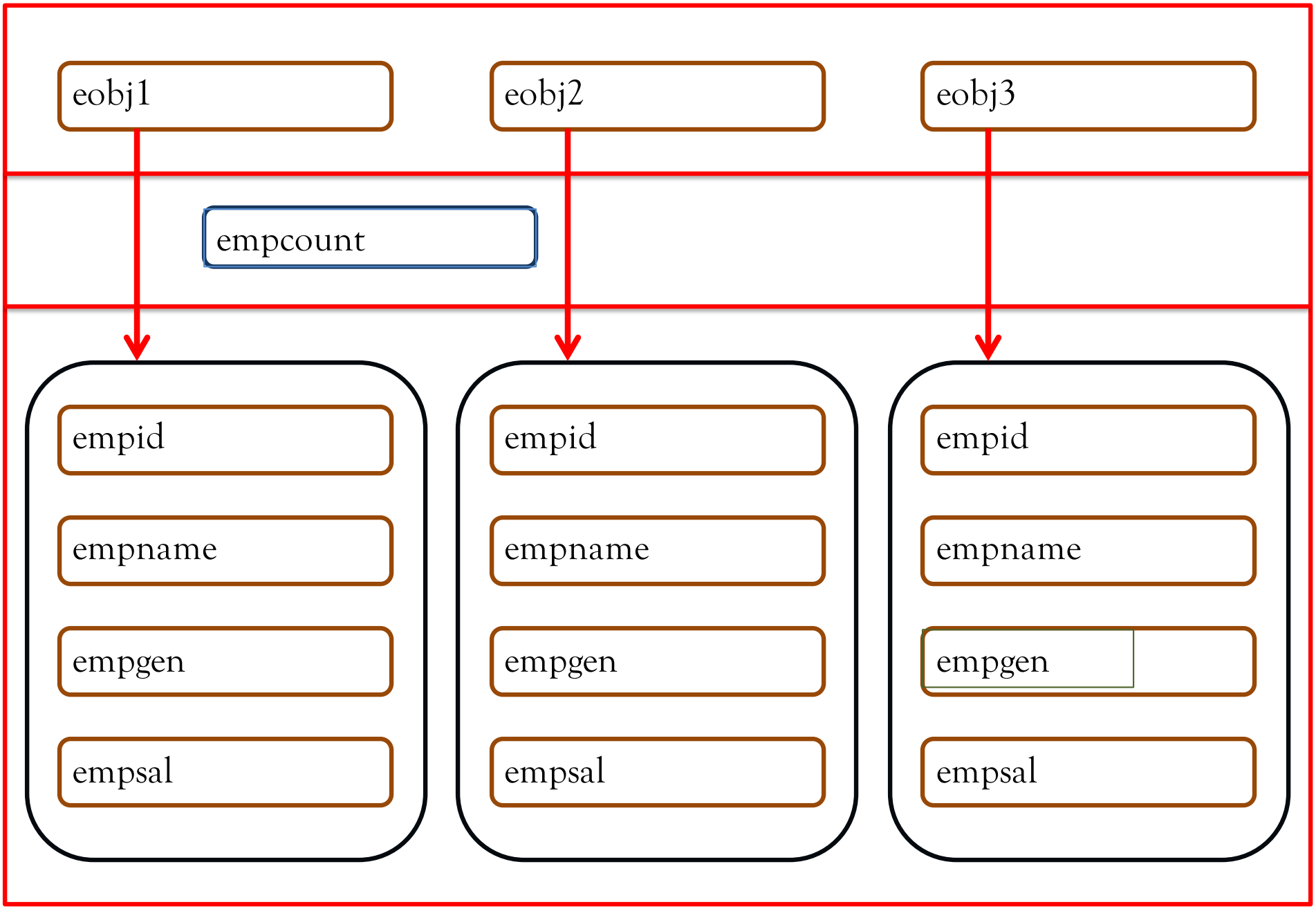
- Consider a class Employee with definition as –
- public class Employee

```
{
    int empid;
    String empname;
    char empgen;
    double empsal;
    int empcount ;
    Employee( )
     {  empcount++;  }
   void input( )  {}
    void output() {}
}
```

contd..


eobj1
eobj2
eobj3
empid
empname
empgen
empsal
empcount
empid
empname
empgen
empsal
empcount
empid
empname
empgen
empsal
empcount

contd..


eobj1
eobj2
eobj3
empcount
empid
empname
empgen
empsal
empid
empname
empgen
empsal
empid
empname
empgen
empsal

contd..

- **Polymorphism**
  - Key feature of OOPs
  - One name many forms
  - Applied in situations where there is commonality of actions
  - Methods also support variable number of arguments.
  - The qualifiers like static, final and access specifiers do not influence polymorphism.
- **Boilerplate code** – boilerplate code is the sections of code that have to be included in many places with no or little alteration.
- The definitions of getter and setter methods can be regarded as boilerplate code. The code is stereotypical in structure and can be generated automatically than written by hand.

contd..

- **Methods with variable arguments**
- Need for methods with variable arguments

```
class Sample
 {
    public int multiply(int a, int b)
       { return a*b; }
    public int multiply(int a, int b, int c)
       { return (a*b)*c; }
     public int multiply(int a, int b, int c, int d)
       { return (a*b)*(c*d); }
   }
```

contd..

- **Variable number of arguments**

- The ... *syntax* tells the compiler that a variable number of arguments will be used, and that these arguments will be stored in the array referred to by a name.
- the variable-length parameter must be the last parameter declared by the method and only one variable-argument parameter is allowed.
- in calling method, the variable argumented method is called with different number of arguments, including no arguments at all; (in such a case the length of the array is zero).
- a method can have "normal" parameters along with a variable-length parameter.

Hash Code

- A **hash code** is an integer value that is associated with each object in **Java** and used to insert and identify an object in a **hash** –based collection.
- The default hashCode() is typically implemented by converting the internal address of the object into an integer.
- Multiple objects can have same hashcode. If **two** objects are **equal** (using the equals() method) then they **have** the **same hashcode**.
- hashCode() is used for bucketing in Hash implementations like HashMap, HashTable, HashSet, etc.
- The value received from hashCode() is used as **bucket** number for storing elements of the Set/Map.
- Its used to facilitate **hashing** in **hash** tables, which are used by data structures like HashMap. This **bucket** number is the address of the element inside the set/map.

equals() method

- **equals**() method equals() method for **content comparison**, is used to compare two objects for the values of the properties.
- equals() evaluates to the comparison of values in the objects.
- if a class does not override the equals() method, then by default it uses equals(Object o) method of the closest parent class that has overridden this method.
- the return value of equals() is **boolean** type.

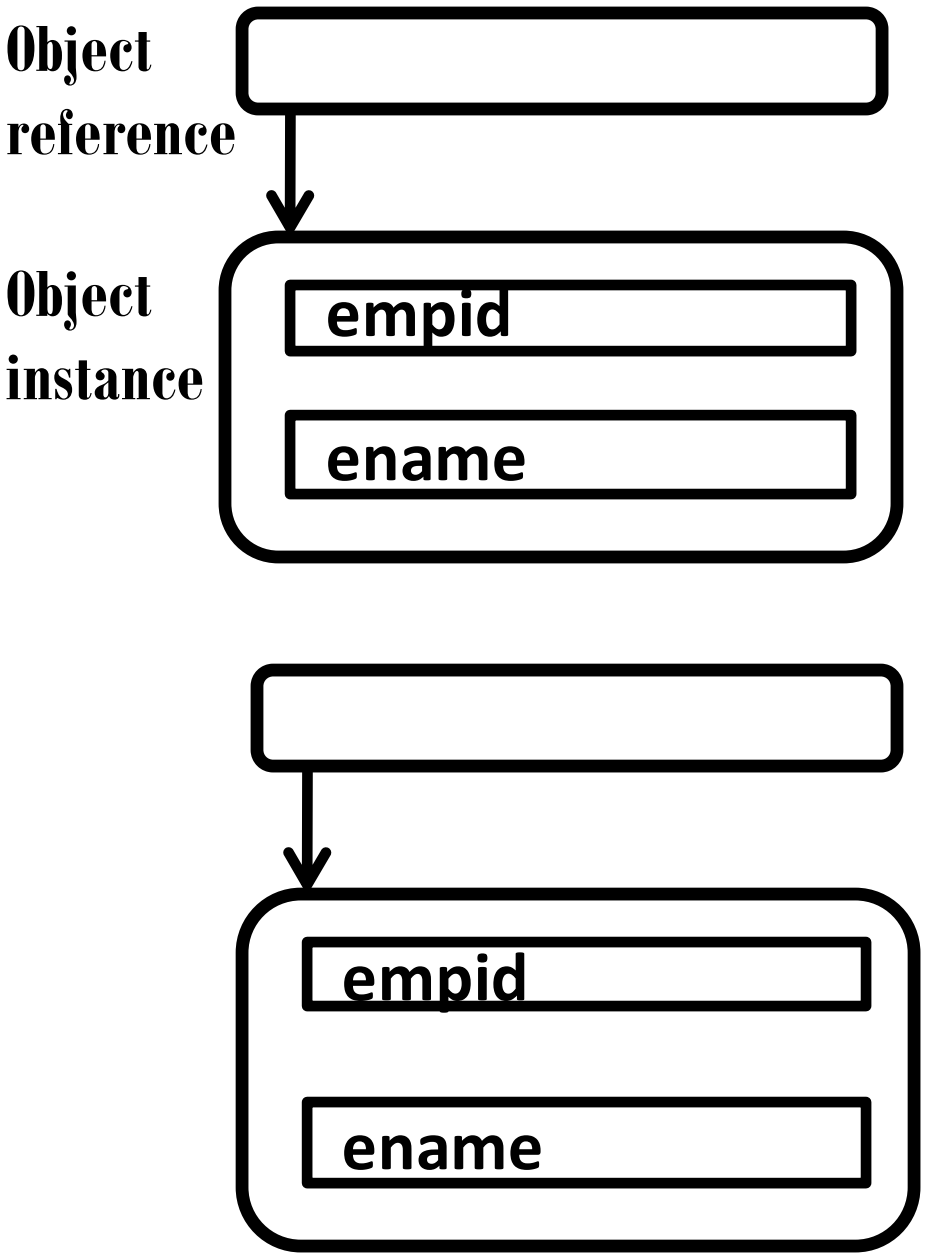
contd..

- **Immutability –** Immutable classes are those class, whose object can not be modified once created, it means any modification on immutable object will result in another immutable object. Objects whose data values does not change. E.g. String
- To make a particular class immutable –
  - All fields of immutable class should be final.
  - Object must be properly constructed i.e. object reference must not leak during construction process.
  - Object should be final in order to restrict sub-class for altering immutability of parent class.

contd..

- Object Array

```
class Employee
{
 int empid;
 String ename;
 void input()
 { empid=101; ename="Akash"; }
 void output()
{ System.out.println(empid);
 System.out.println(ename); }
}
public static void main(String[] a)
{  Employee eobj[];
   eobj = new Employee[4];
  }
```



contd..

- **Nested class –** Class can be a member of another class. Such a class is called as Nested class.
  - Nested classes are of two types – **static** and **non static**.
  - The non static nested class is called as **Inner class**.
- **Anonymous class –** An inner class can be anonymous.
  - Anonymous class –
  - Defined at location where
  - Combine the process of definition and instantiation in single step.

contd..

- **public class Employee**

```
{ private int empid;
  private  String empname;
  public void setEmpid(int empid) { this.empid= empid;}
  public int getEmpid() { return empid; }
  public void setEmpname(String empname) { ……..}
  public String getEmpname() { return empname; }
}
```

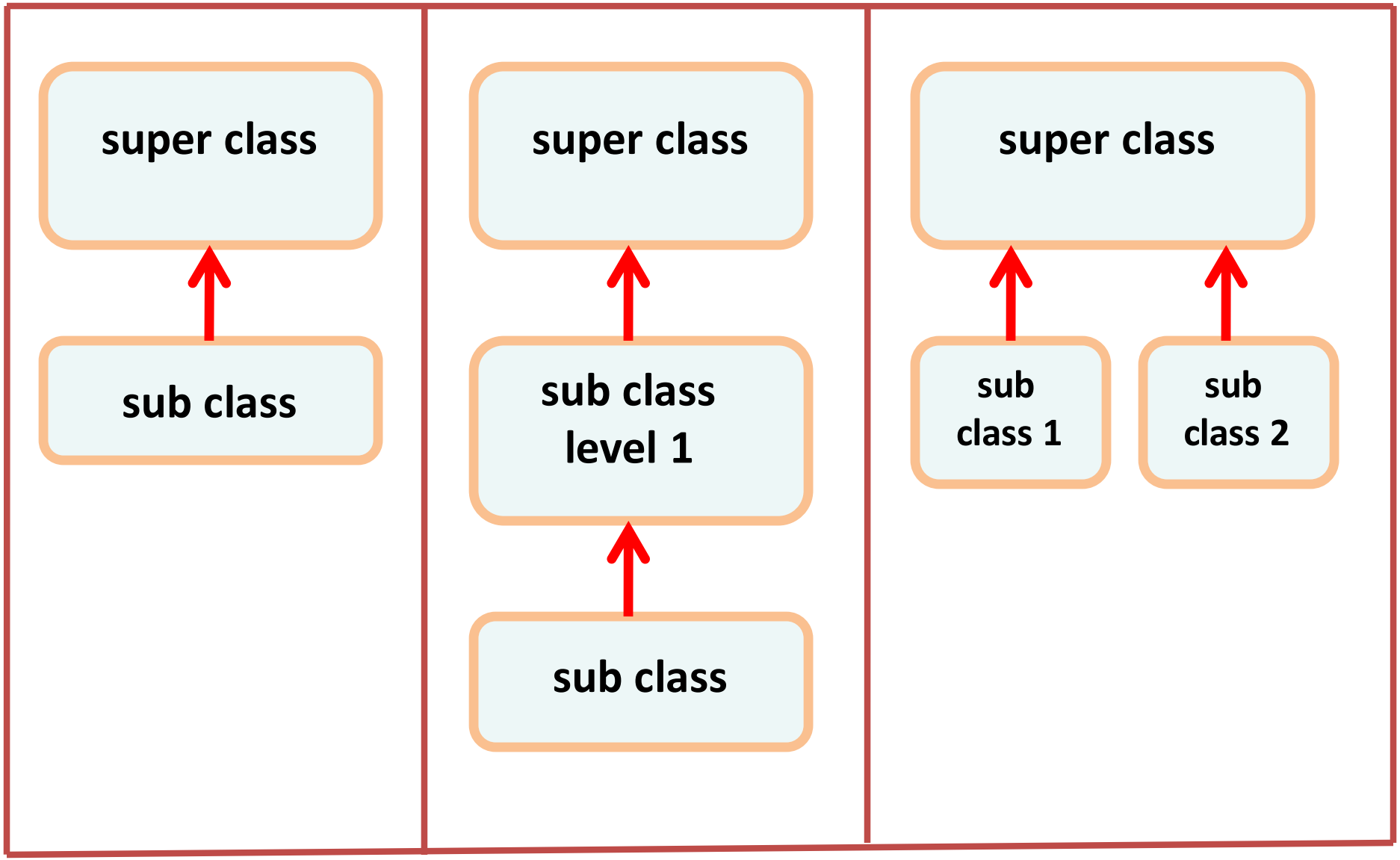
- The only purpose of this class is to silently accept any value of the proper types.
- To make it constrained, a vetoable event needs to be produced and distributed to the consumers.

Inheritance

- Inheritance concept
- Inheritance establishes relationship
  - has – a
  - is – a
- Application of inheritance
  - organizational hierarchy
  - role plays
  - e.g. Hierarchical relations
    - ✓ Head office – branch office
    - ✓ Team leader – Team members
    - ✓ Vertical relations

contd..

- Inheritance types supported by Java

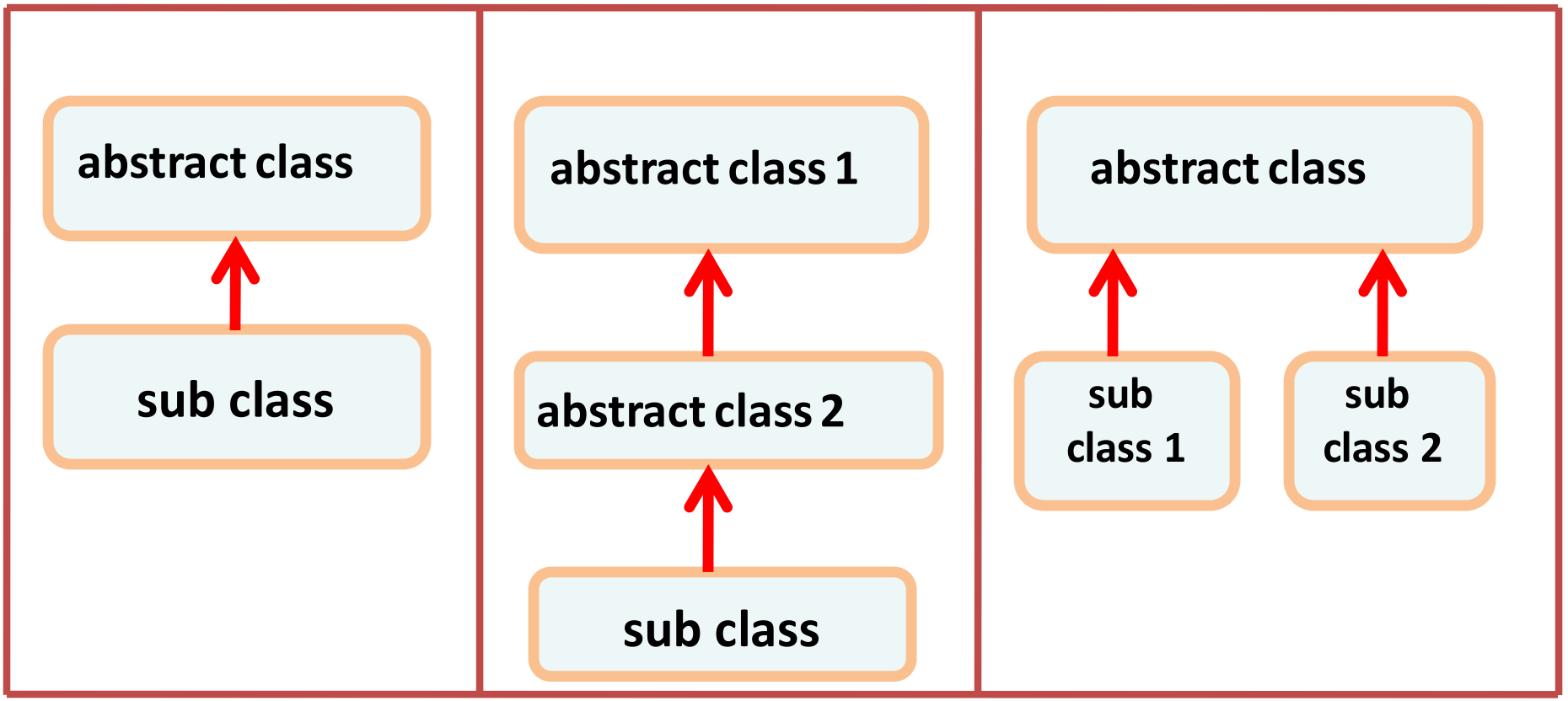


contd..

- Issues in inheritance
  - **Access specifiers** – public, protected, default, private.
  - Behavior of constructors
    - ✓ Constructors are executed in superclass – subclass sequence
    - ✓ Abstract classes can have constructors
    - ✓ Usage of keyword – **super**
  - **Qualifiers** associated with inheritance – super, final, static, volatile, transient.
  - Behavior of garbage collectors.
  - Implementing ***polymorphism*** with inheritance.
  - Object type casting with inheritance.

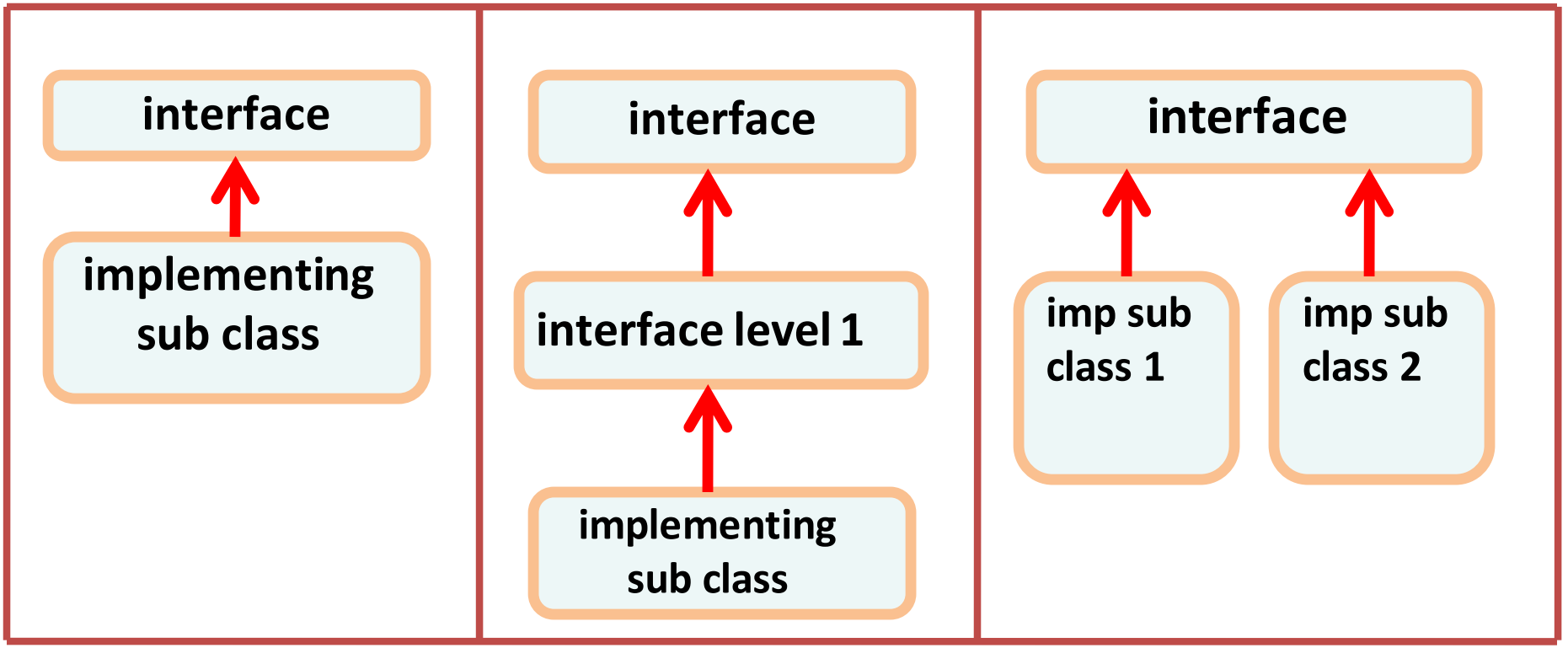
abstract class

- An **abstract class** is a **class** which cannot be instantiated. The **purpose** of an **abstract class** is to function as a base for subclasses.
- **Abstract class** defines some common behavior that can be inherited by multiple subclasses, without implementing the entire **class**.
- Provides for separation of ***definition*** and ***implementation***.



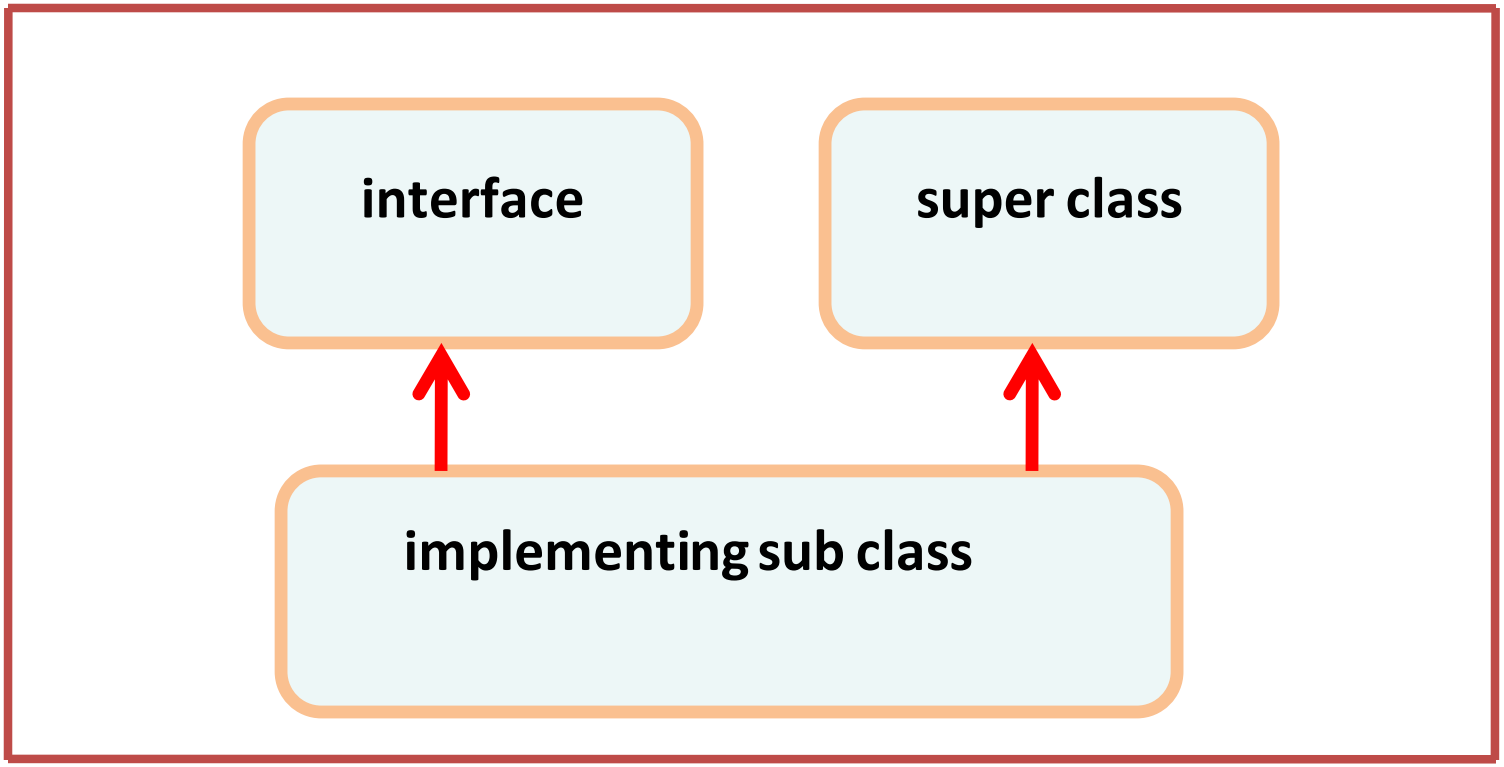
interface

- Pure abstract class – **interface**



contd..

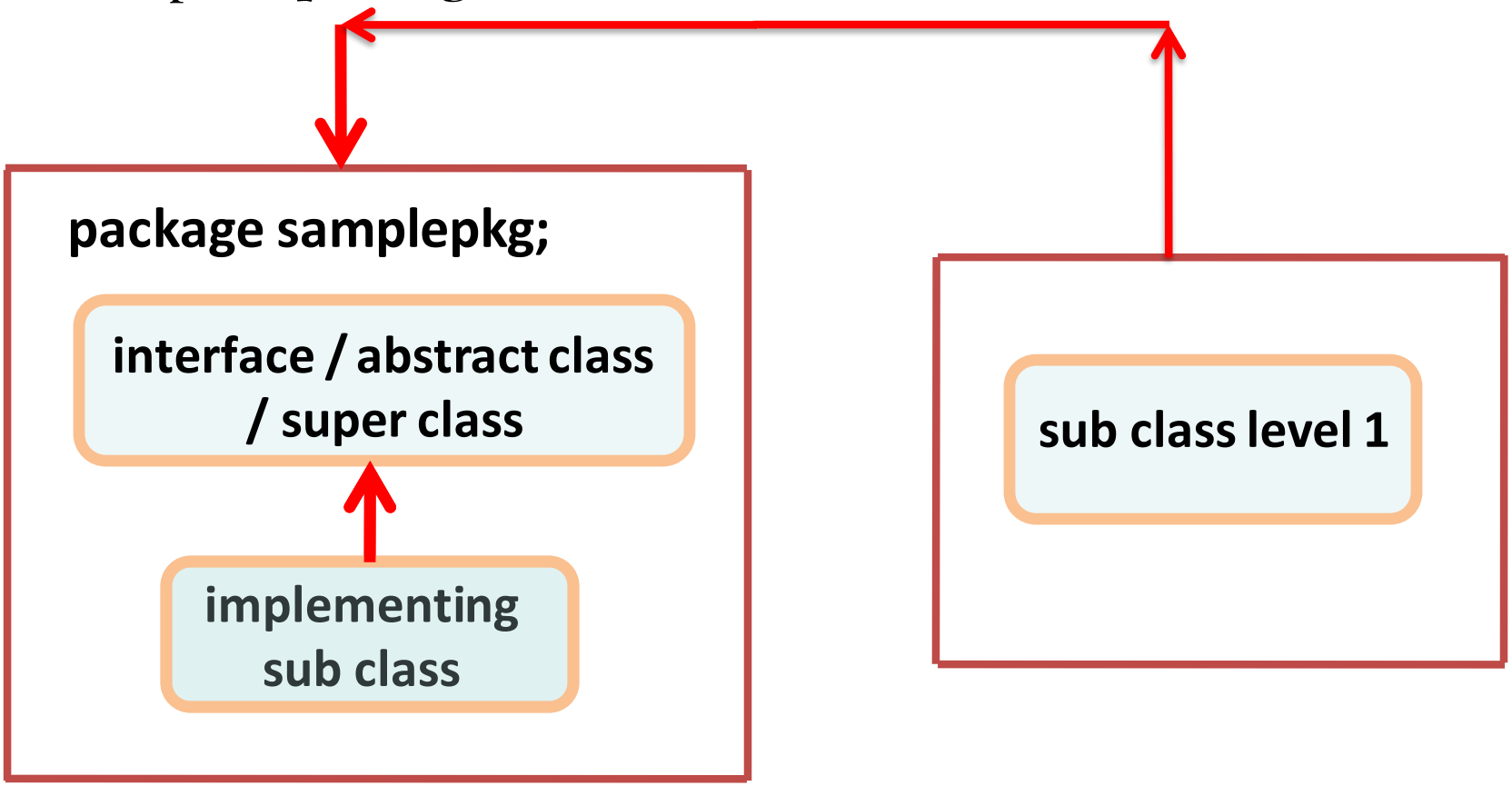
- Use of interface – **multiple inheritance**



- Ambiguity taken care of in the above arrangement.

package

- Concept of ***package***



Accessibility rules

	Private	Default	Protected	Public
Within Class				
Outside Class Same package				
Subclass Same package				
Outside class Different package				
Sub class Different package				

static inheritance

- Inheritance of static members –
  - static members are not inherited.
  - Super class static members can be accessed using subclass name.
- Upcasting and Downcasting
  - Upcasting is assigning subclass object to super class ref.
  - Downcasting is assigning superclass object to sub class ref.

default methods & functional interface

- **default methods** – Methods in interface can be provided an implementation. Such method are called as default methods.

  An interface can have any number of default methods.

  The declaration of default methods is as follows.

  public default type method–name( )

  { //code.....

  }

- **Functional Interface –**
  - An interface with **only single abstract method** is called functional interface(or Single Abstract method interface), for example: Runnable, Callable, ActionListener etc.
  - **To use functional interface:**
    Pre Java 8: anonymous inner classes.
    Post Java 8: lambda expressions.

Lambda Expressions

- **Lambda Expressions**

- A lambda expression is an anonymous function, that doesn't belong to any class.

- A lambda expression in Java has the following parts
  Lambda expression **only has body and parameter list**.
  1. **No** name – function is anonymous so we don't care about the name
  2. Parameter list
  3. Body – This is the main part of the function.
  4. **No** return type – The Java 8 compiler is able to infer the return type by checking the code, need not be mentioned explicitly.

- To create a lambda expression, specify input parameters (if there are any) on the left side of the lambda operator –>, and place the expression or block of statements on the right side of lambda operator. E.g. (x, y) –> x + y

Method referencing

- **Method Referencing**
- A method reference refers to a method of functional interface without invoking it; the :: operator is used to separate the class or object from the method name
- Constructor reference is similarly used to refer to a constructor without creating a new instance of the named class or array type.
- In a **method reference**, the object (or class) that contains the **method** that is to be called is placed before the delimiter :: operator and the name of the **method is** provided after it without arguments. Object :: methodName
- Examples of method and constructor references

  System::getProperty
  System.out::println
  "abc"::length
  ArrayList::new int[]::new
- Its basically a shortcut for lambdas.

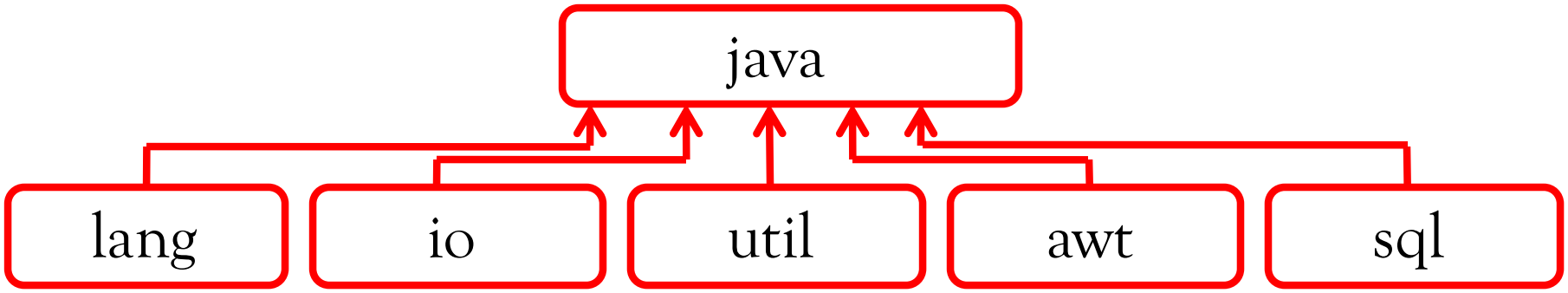
contd..

- **Four types of method references**

1. Method reference to an instance method of an object – object::instanceMethod
2. Method reference to a static method of a class – Class::staticMethod
3. Method reference to an instance method of an arbitrary object of a particular type – Class::instanceMethod
4. Method reference to a constructor – Class::new

which can let you reuse existing method definitions and pass them just like lambda expressions.

packages in java library



java.lang

- **java.lang** provides most basic utilities for Java programming. The package is imported by default into each of the Java program.
- **Primitive type wrappers** – Java provides classes that correspond to each of the primitive types. These classes encapsulate or wrap the primitive types within a class, hence referred as type wrappers or Wrapper classes.
- **Number** – An abstract class, containing abstract methods that return the value of the object in each of the different number formats. The methods are –

(i) byte byteValue() ; (ii) double doubleValue() ;

(iii) float floatValue() ; (iv) int intValue();

(v) long longValue() ; (vi) short shortValue() ;

contd..

- Number has six concrete subclasses that hold explicit values of each numeric type.

Sl.No.	Class	Constructors
1.	Byte	(i) Byte(byte num) (ii) Byte(String s) throws NumberFormatException
2.	Short	(i) Short(short num) (ii) Short(String s) throws NumberFormatException
3.	Integer	(i) Integer(int num) (ii) Integer(String s) throws NumberFormatException
4.	Long	(i) Long(long num) (ii) Long(String s) throws NumberFormatException
5.	Float	(i) Float(double num) (ii) Float(float num) (iii) Float(String s) throws NumberFormatException
6.	Double	(i) Double(double num) (ii) Double(String s) throws NumberFormatException

contd..

Important classes of java.lang

1. Class
2. Classloader
3. Math
4. Object
5. Package
6. Throwable
7. Thread
8. String
9. StringBuffer
10. System
11. Void

Description of java.lang classes

1. **Class** – Class encapsulates the run time state of an object or interface.
   - Objects of the Class are created automatically when Class is loaded, i.e. Class object cannot be explicitly declared.
   - Class is a generic type and is declared as
     class Class <T> // T is any class or interface.
   - Many methods of Class are used in annotation programming

- **Methods** –
  - **getClassLoader()** – returns a class loader object for each of public classes and interfaces that are members of invoking object.
  - **Class<?> for Name(String name)** – returns a class object given its complete name.

contd..

2. **ClassLoader** – An abstract class, it defines how classes are loaded. An application creates a subclass of ClassLoader whereby classes can be loaded in a way other than the way Java's run-time system does.

3. **Math** – Math class contains all floating-point functions that are used for geometry, trigonometry, and general purpose.

- **Methods** –
  - static double cbrt(double arg)
  - static double exp(double arg)
  - static double pow(double x, double y)
  - static double sqrt(double arg)
  - static double log(double arg)
  - static double log10(double arg)

contd..

4. **Object** – Object is the superclass of all other classes. It means that a ref - variable of type Object can refer to an object of any other class. As arrays are implemented as class, variable of type Object can also refer to any array.

- **Methods of Object class** –
  a) Object clone() throws CloneNotSupportedException – Creates a new object that is the same as the invoking object.
  b) boolean equals(Object ob) – Returns true if the invoking object is equivalent to ob.
  c) void finalize() throws Throwable – the default finalize method usually overridden by subclasses.
  d) final Class getClass() – gets a Class object that describes the invoking object.
  e) int hashCode() – returns the hash code associated with the object.

contd..

f) String toString() – returns a string that describes the object.

g) final void notify() – resumes execution of a thread waiting on the invoking object.

h) final void notifyAll() – resumes execution of all threads waiting on invoking object.

i) final void wait() – waits on another thread of execution.

j) final void wait(long milliseconds) throws InterruptedException waits on another thread of execution for a specified number of milliseconds.

k) final void wait(long milliseconds, int nanoseconds) throws InterruptedException – waits on another thread of execution for a specified number of milliseconds plus nanoseconds.

Interfaces in java.lang

1. **Runnable** – An interface which a class must implement to initiate a separate thread of execution. Runnable defines only one abstract method, called run(). Its defined as

   void run().

2. **Cloneable** – Cloneable interface defines no members. It is used to indicate that a class allows bitwise copy of an object to be made. If the method clone() (of Object class) is called on class that does not implement Cloneable, then CloneNotSupportedException is thrown.
3. **Comparable** – The interface Comparable is generic. The objects of classes that implement Comparable can be ordered. The interface declares one method –

   int compareTo(T obj)

Assertion

- **Assertion –** a condition that should be true during program execution. e.g. there is a method that should always return a positive integer value. At run time, if the condition actually is true, no other action takes place, else an **AssertionError** is thrown.
- **Assertions** are often used during testing to verify that some expected condition is actually met. They are not usually used for released code.
- assert – used to test an assertion. **assert** keyword has two forms.
- **a.assert condition ;** *condition* is an expression that must evaluate to a Boolean result.
- **b.assert condition : expr ;** *expr* is a value that is passed to the **AssertionError**

Exceptions

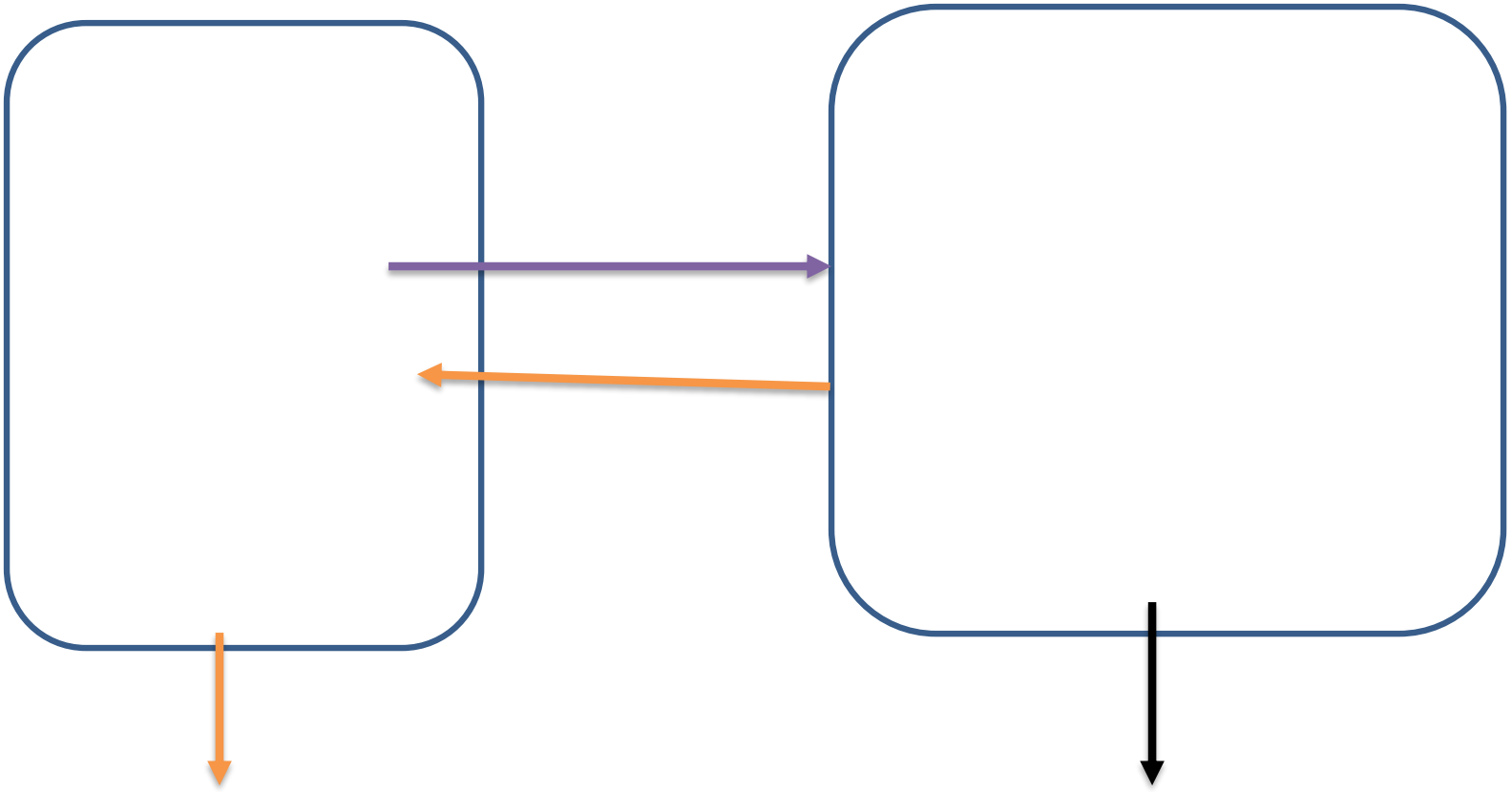
Exceptions

- Concept – Exceptions are run time occurrences of the piece of code which help system to be robust.
- An *exception* is an *abnormal* condition that arises in a code sequence at run time.
- When an exceptional condition arises, an object representing that exception is created and *thrown in the method that caused the error. That method may* choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught and processed.*
- Exceptions are runtime and occur from situations beyond control of developer. They are of two types –

  a. Checked exceptions b. Unchecked exceptions
- Errors are compile time and are mainly semantic in nature.

main(String args[])

ArrayIndexOutOfBoundsException



contd..

- Exception handling keywords.

  try ; catch ; throw ; throws ; finally.

- The try, catch and finally form blocks of code with local scope.

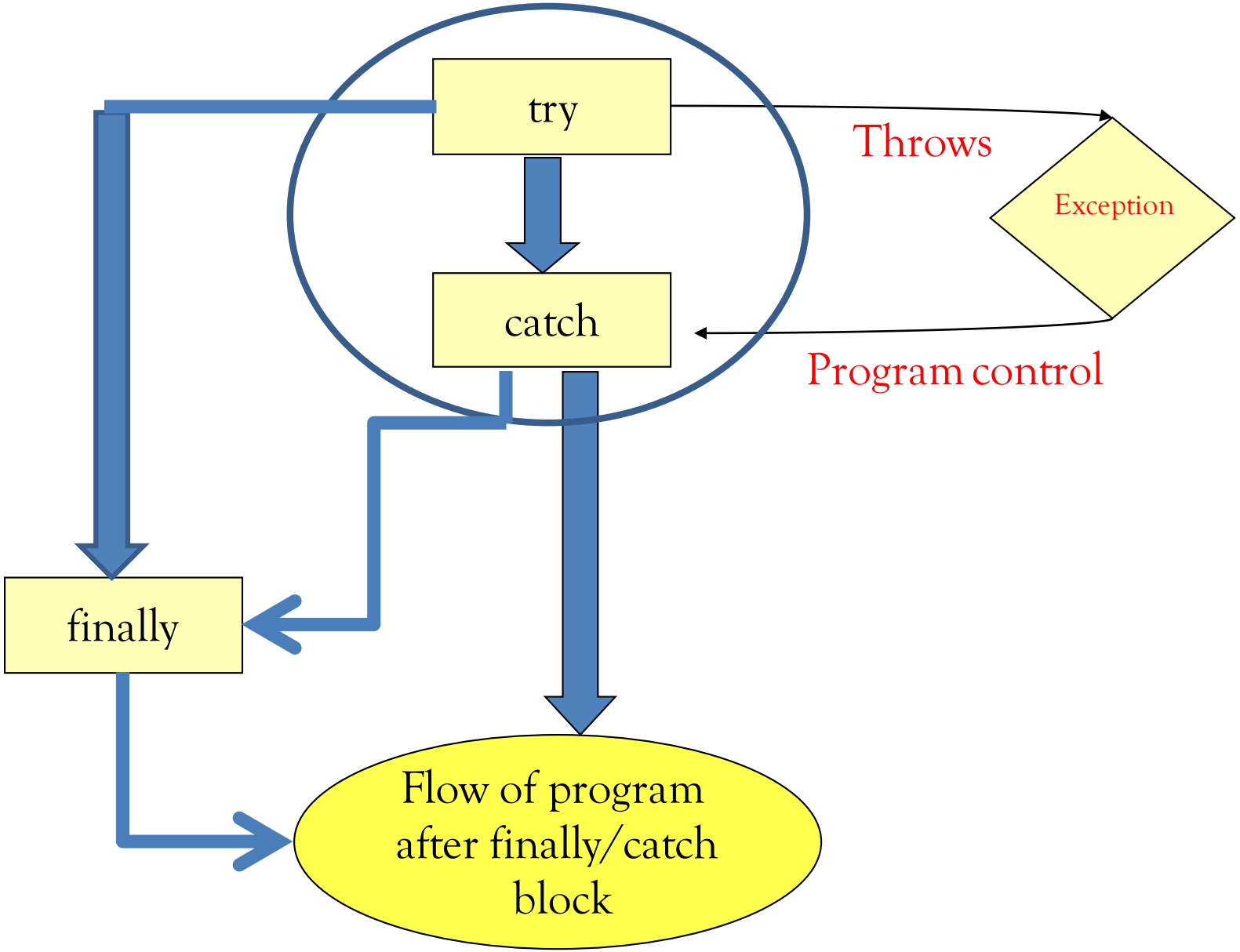
- **try block** – Program statements monitor for exceptions are contained within a try block.

- **catch block –** If an exception occurs within the try block, it is thrown. The code catches this exception (using catch) and handle it in some rational manner.

- **finally –** Any code that must be executed mandatorily after a try block completes is put in a finally block.

Note : One try block can be followed with multiple catch blocks. The exception will look for matching catch block.

try, catch, finally blocks

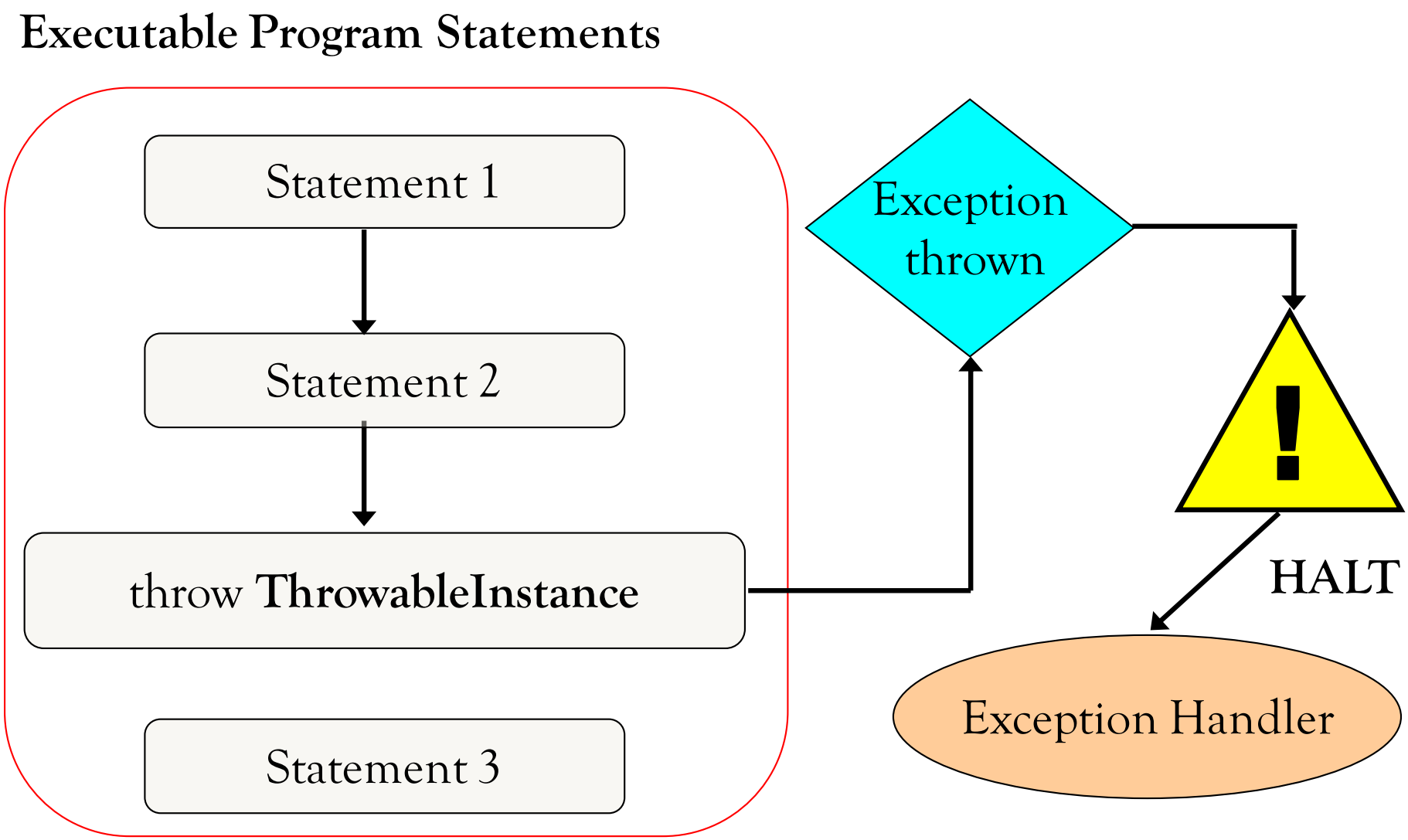


throw, throws

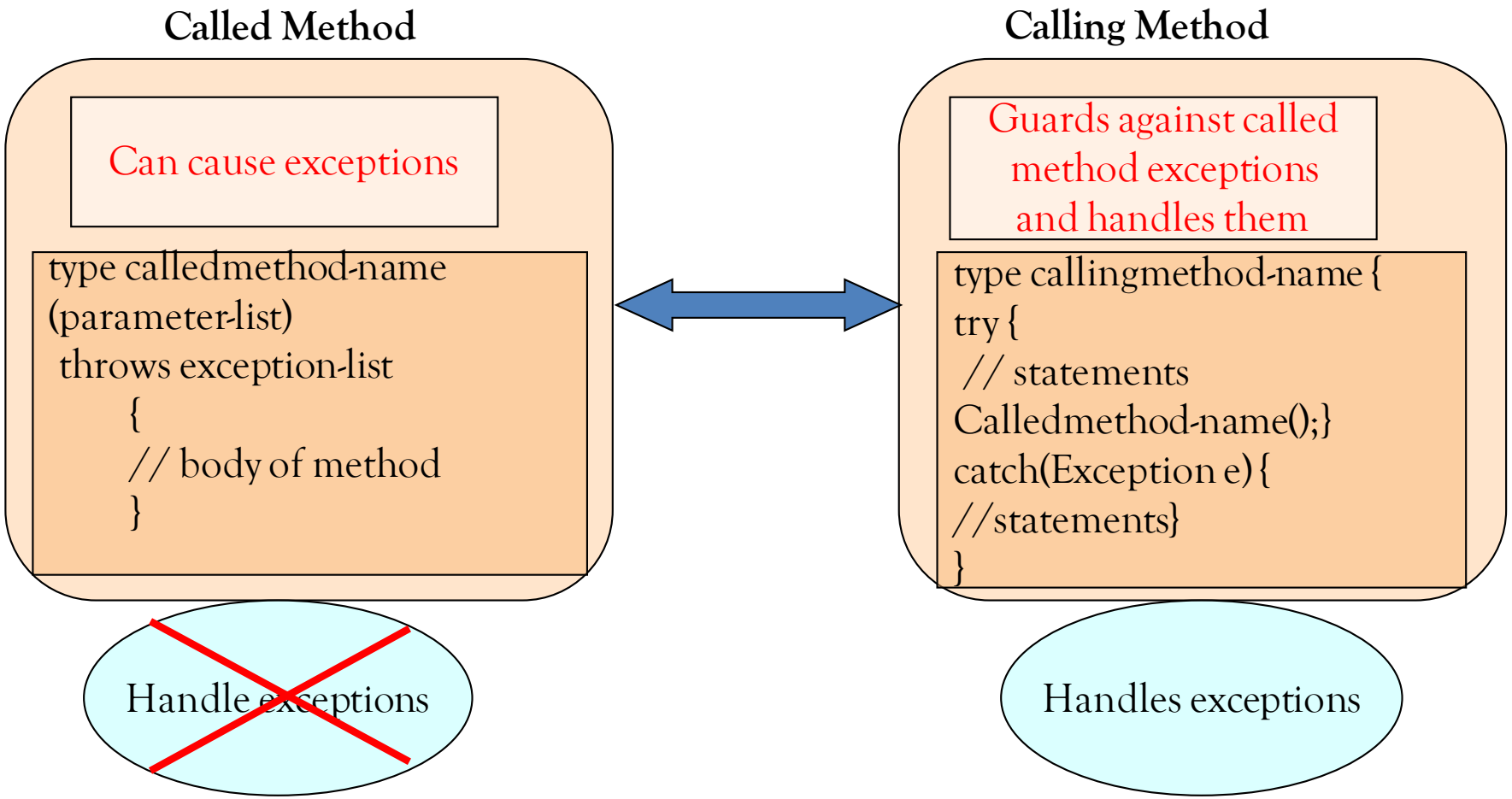
- **throw** – used in situation where explicit throwing of exception happens. To manually throw an exception.
- The flow of execution stops immediately after the throw statement; any subsequent statements are not executed.
- **throws** – Any exception that is thrown out of a method must be specified as such by a throws clause.

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

contd..

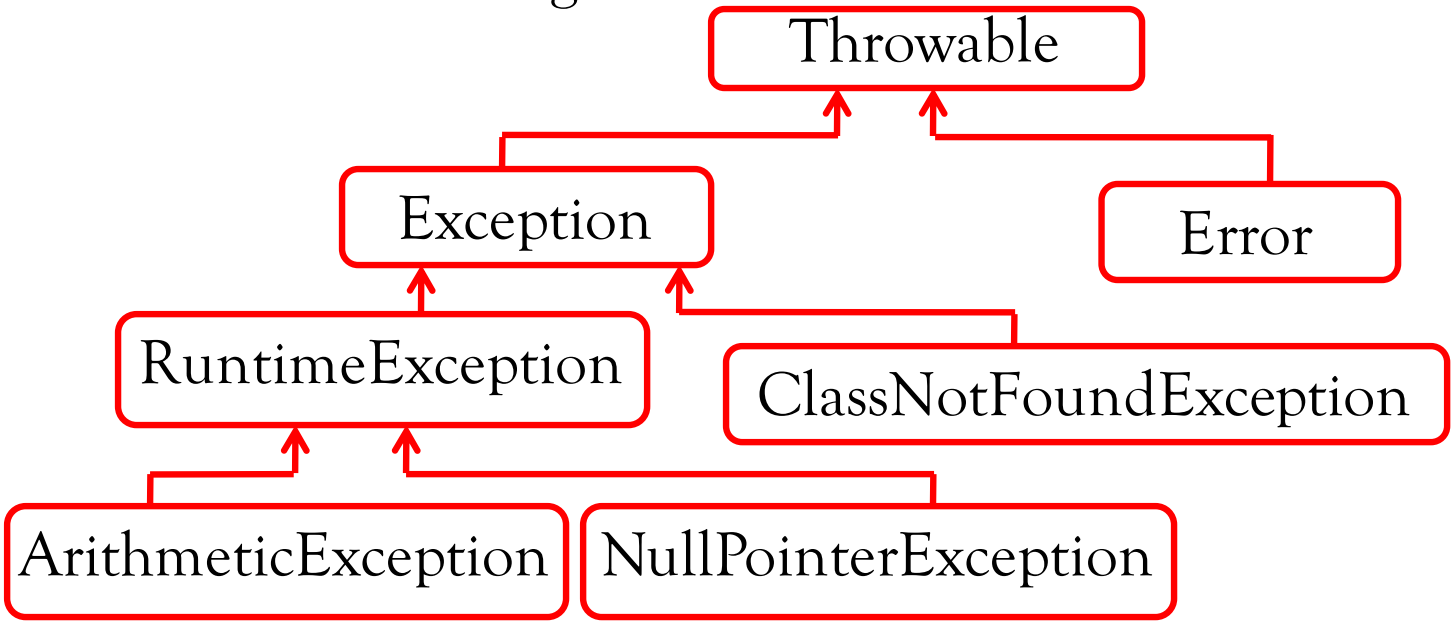

Executable Program Statements
Statement 1
Statement 2
throw ThrowableInstance
Statement 3
Exception thrown
!
HALT
Exception Handler

contd..


Called Method
Calling Method
Can cause exceptions
Guards against called method exceptions and handles them
type calledmethod-name (parameter-list) throws exception-list { // body of method }
type callingmethod-name { try { // statements Calledmethod-name();} catch(Exception e){ //statements} }
Handle exceptions
Handles exceptions

contd..

- Exception API – All exception types are subclasses of the built – in class **Throwable, having** two subclasses viz.. **Exception** and **Error.**
- **Exception** is used for situations that user programs should catch and to create custom exceptions.
- **Error** defines exceptions that are not expected to be caught under normal circumstances by user program. Used by the JRE system to indicate errors E.g. Stack overflow.



contd..

- Java defines several exception classes.
- The most general of these exceptions are subclasses of the standard type RuntimeException. These exceptions need not be included in any method's throws list. They are called *unchecked exceptions because the compiler does not check to see if a method handles or* throws these exceptions.
- The other category of exceptions that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions.*
- Java defines several other types of exceptions that relate to its various class libraries.

Exception handling from jdk 1.7

- **Multi catch statement –** Prior to Java 7, multiple catch blocks had to be written for the same try block. Java 7 onwards multiple exceptions handled in same manner can be combined single catch block separated by | operator.
- **final rethrow :** Allows you to catch an exception type and it's subtype and rethrow it without having to add a throws clause to the method signature.

```
try { // some code }
    catch (final Throwable ex)
{ // some more code throw ex; }
```

Using the final keyword it allows you to throw an exception of the exact dynamic type that will be thrown. So if an IOException occurs, an IOException will be thrown.

Multi Threading

Concepts

- **Execution Modes**
  - **Serial**
  - **Concurrent**
  - **Parallel**

Sl. No.	Processed Based Execution	Thread Based Execution
1	Processes are heavy weight	Light weight
2	Long response time	Short response time
3	Context switching is difficult	Context switching made easy
4		Multithreading replaces event loop programming by dividing tasks into discrete logical units.

contd..

- Threads allow multiple streams of program control flow to coexist within a process.
- They share process–wide resources such as **memory** and **file handlers**, but have their own **program counter**, **stack**, and **local variables**.
- Threads also provide a natural decomposition for exploiting hardware parallelism on multiprocessor systems.
- Multiple threads within the same program can be scheduled simultaneously on multiple CPUs.
- Threads are sometimes called lightweight processes, and most modern operating systems treat threads, not processes, as the basic units of scheduling.
- In the absence of explicit coordination, threads execute simultaneously and asynchronously with respect to one another.

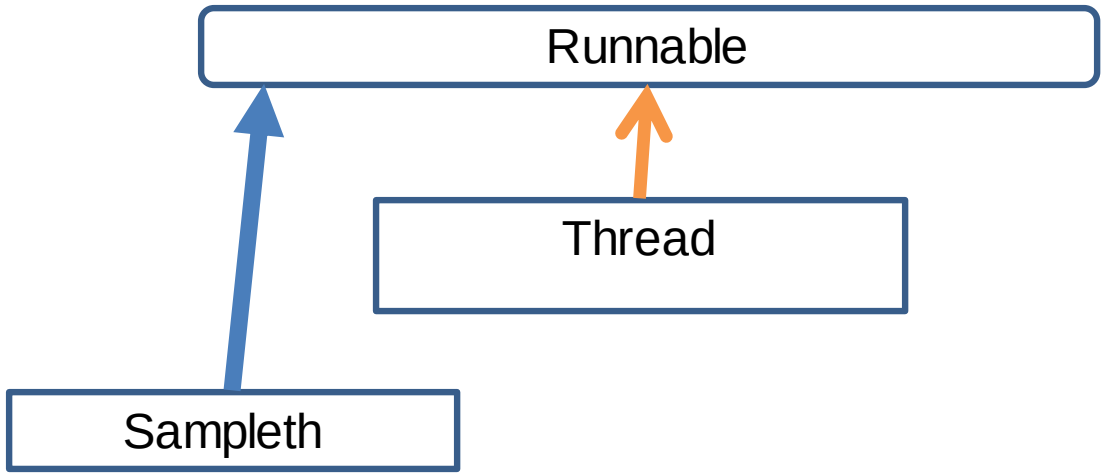
contd..

- Threads are objects in execution. More than 1 objects get executed concurrently.
- Threads share the memory address space of their owning process,
  - ✓ all threads within a process have access to the same variables and allocate objects from the same heap, which allows finer–grained data sharing than inter–process mechanisms.
  - ✓ a thread may modify variables that another thread is in the middle of using, with unpredictable results.
- A class is thread–safe if it behaves correctly when accessed from multiple threads.
- Stateless objects are always thread-safe.
- Race condition
- Visibility

contd..

- Two ways to **create threads** –
  - Extend the **Thread** class
  - Implement the **Runnable** interface
- The **constructors** of Thread class are –
  1. Thread()
  2. Thread(String threadName)
  3. Thread(Runnable threadob)
  4. Thread(Runnable threadob, String threadName)
  5. Thread(ThreadGroup groupObj, Runnable threadob)
  6. Thread(ThreadGroup groupObj, Runnable threadob, String threadName)

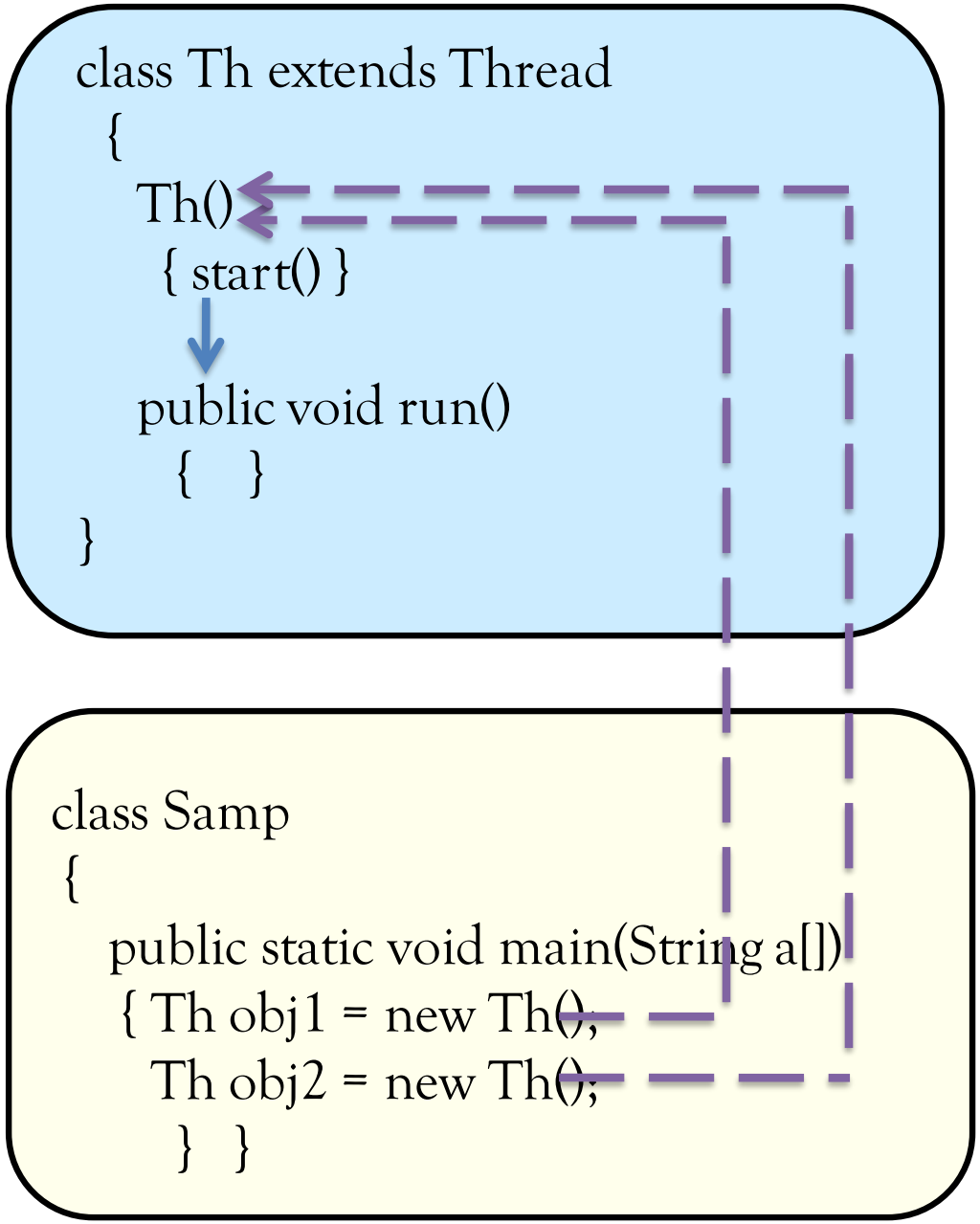
Note : **groupObj** is the thread group to which new thread will belong; default is the group of **parent thread**. If **thread name** is not specified, **JVM** gives a name.


Runnable
Thread
Sampleth

contd..

- **Execution sequence**

- **main** is the calling thread from which all child threads (thread objects) are created or declared and called.
- Threads are initialized in the child class constructor by the method **start**().
- **start**() invokes **run**(). The thread code is implemented in the **run**().
- Reordering



contd..

- **Methods of Thread class** –

- **static Thread currentThread( )** returns a Thread object that encapsulates the thread that calls this method.
- **void destroy( )** terminates the thread.
- **final String getName( )** returns the thread's name.
- **final int getPriority( )** returns the thread's priority setting.
- **join methods**
  - ✓ **final void join( )** throws IE waits until the thread terminates.
  - ✓ **final void join(long *ms*)** throws IE waits up to the specified number of milliseconds for the thread on which it is called to terminate.
  - ✓ **final void join(long *ms*, int *ns*)** throws IE waits up to the specified number of milliseconds plus nanoseconds for the thread on which it is called to terminate.

contd..

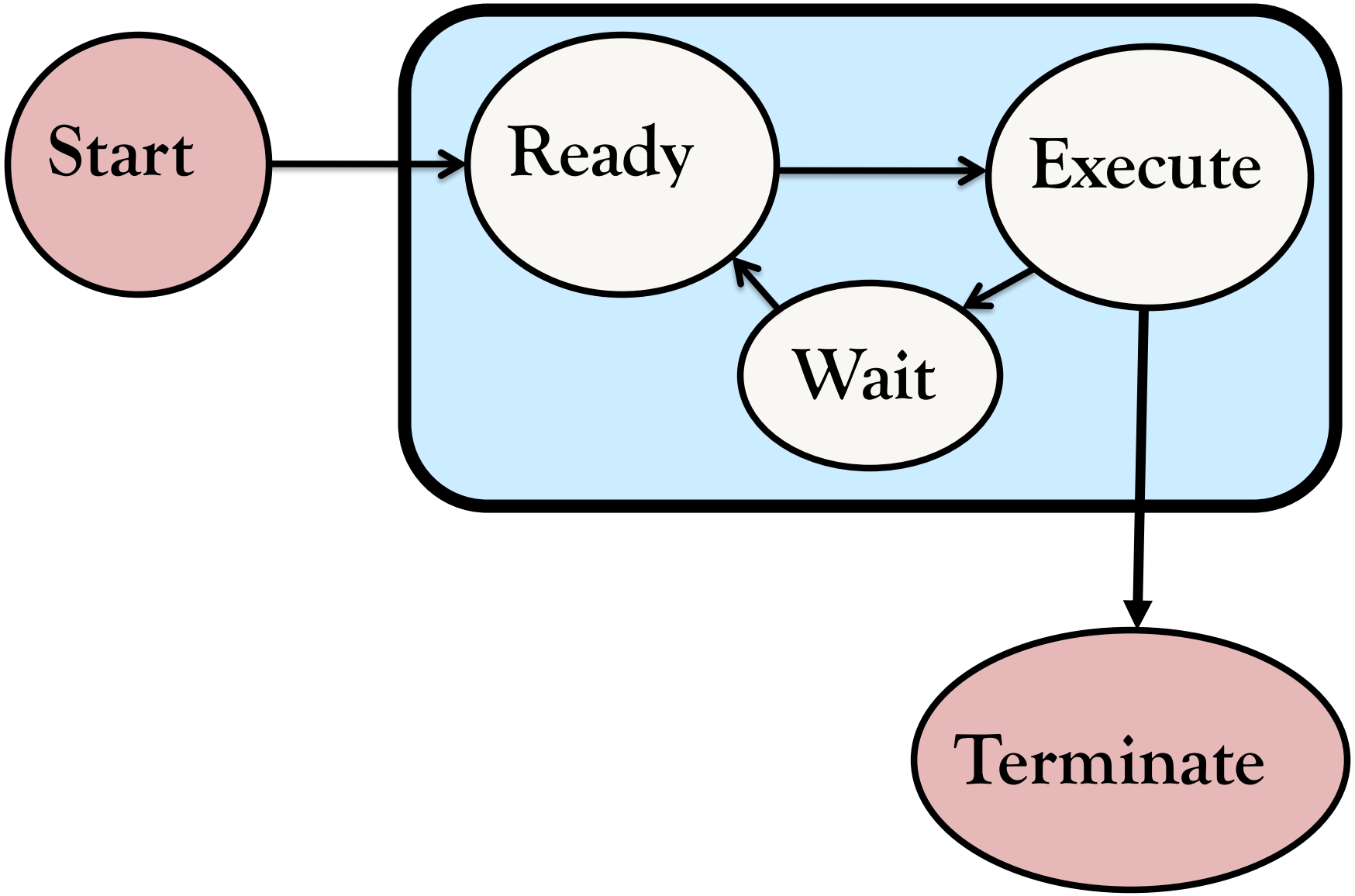
- **final boolean isAlive( )** returns true if the thread is still active else it returns false.
- **void run( )** begins execution of a thread.
- **final void setName(String *threadName*)** sets the name of the thread to that specified by *threadName*.
- **final void setPriority(int *priority*)** sets the priority of the thread to that specified by *priority*.
- **static void sleep(long *milliseconds*)** throws InterruptedException suspends execution of the thread for the specified number of milliseconds.
- **static void sleep(long *milliseconds*,int *nanoseconds*)** throws InterruptedException suspends execution of the thread for the specified number of milliseconds plus nanoseconds.
- **void start( )** starts execution of the thread.
- **String toString( )** returns the string equivalent of a thread.
- **static void yield( )** the calling thread yields the CPU to another thread.

contd..

- **Thread Priority** –

- The amount of CPU time that a thread gets depends on **priority** and other factors. Its O.S. influenced.
  - ✓ A higher priority thread can preempt a lower priority one.
  - ✓ Threads of equal priority get equal CPU time in round-robin fashion, in non-preemptive O.S.
  - ✓ Each thread does encounter a blocking condition.
- Thread class defines three **final** variables **NORM_PRIORITY, MIN_PRIORITY** and **MAX_PRIORITY** which provide the priority level ranges a thread can take up.

Life Cycle of a Thread



contd..

- **Synchronization** –
  - A methodology which ensures that the accessed resource will be used by only one thread at a time.
  - Synchronization aims to prevent **race condition**.
  - Java implements synchronization through its own language and not through O.S. primitives.
  - Keyword – **synchronized**.
  - While a thread is inside a synchronized method, all other threads that try to call it (or any other sync method) on the same instance **will have to wait**.

contd..

- To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the synchronized method simply returns from the synchronized method.
- Its also possible to put calls to the methods defined by a class inside a **synchronized** block.
- synchronized(*object*)

```
{
    // statements to be synchronized
}
```

Strings

Strings

- Strings are **immutable** objects belonging to the class String.
- String class is defined in **java.lang** and declared as **final**.
- String class constructors –
  - String(char chars[]);
  - String(char chars[], int startindex, int numchars);
  - String(String obj);
- String length the method used is **length**();
- Java strings begin and end on the same line. There is no line-continuation or escape sequence.

contd..

- A class can be **final** but also **mutable**.

Note : **final** and **immutable** are two different issues.

- The **immutability** feature is useful in these situations –
  - To make things efficient – **i.e.** less space and more performance, allows this object to be shared efficiently without fear of having the underlying value change.
  - When loading a class in Classloader, Ipaddress, username, password, Database URL are all passed as String
- The qualifier **final** is useful in following situations –
  - String can be used as key in HashMap and Hashtable.
  - The user cannot override any of its methods but just have to use them.

String Operations

- Concatenation – using the operator +. Concatenation of String types and String with other types.
- Character Extraction –
  - char charAt(int where);
  - void getChars(int sourceStart, int sourceEnd, char target[], int targetStart);
  - byte[] getBytes()
  - char[] toCharArray()
- String Comparison –
  - boolean equals()
  - boolean equalsIgnoreCase()
  - equals and ==
  - int compareTo(String str)

contd..

- Searching Strings –
  - int indexOf(char ch)
  - int lastIndexOf(char ch)
  - int indexOf(String s)
  - int lastIndexOf(String s)
  - int indexOf(char ch, int startIndex)
  - int lastIndexOf(char ch, int startIndex)
  - int indexOf(String s, int startIndex)
  - int lastIndexOf(String s, int startIndex)

contd..

- Modifying the String –
  - String substring(int startIndex)
  - String substring(int startIndex, int endIndex)
  - String concat(String str)
  - String replace(char original, char replacement)
  - String trim();
- Changing the case of characters
  - String toLowerCase()
  - String toUpperCase()
- valueOf() – converts data from its internal format into a human readable format. Applied in situations when a String representation of some other data type is needed.
- Note – String class has no methods which can change the state of the String object.

StringBuffer

- Provides for mutable String objects
- The following are the methods of StringBuffer.

- Append
  - StringBuffer append(String str)
  - StringBuffer append(int num)
  - StringBuffer append(Object obj)
- Insert
  - StringBuffer insert(int index, String s)
  - StringBuffer insert(int index, char ch)
  - StringBuffer insert(int index, Object obj)
- Reverse
  - StringBuffer reverse()

contd..

- Delete
  - StringBuffer delete(int startIndex, int endIndex)
  - StringBuffer deleteCharAt(int loc)
- Replace
  - StringBuffer replace(int startIndex, int endIndex, String str)
- SubString
  - String substring(int startIndex)
  - String substring(int startIndex, int endIndex)

StringBuilder

- Introduced from jdk 1.5
- Provides for mutable operations on Strings.
- Main operations are append and insert.
- StringBuilder is unsynchronized whereas StringBuffer is synchronized. i.e. StringBuffer is thread safe.

Annotation

Annotations

- Annotation is supplementary information embedded into the source file, which is used by various tools during development and deployment.
  - provides for compiler generated boilerplate code.
  - do not change the actions of the program.
  - ensure the consistency between classes,
  - can check the validity of the parameters passed by the clients at run-time.
- Annotation is created using the interface based mechanism. Annotations consist of method declarations, which act much as data members.

```
@ Retention(RetentionPolicy.RUNTIME)
@ interface  exampleanno
        {  String str();
        int val(); }
```

contd..

- When an annotation is applied the members are given the values.
  - @exampleanno (str = “CoreJava”, val = 25)

    public static void samp()

    { .....

    .....}
- An annotation can annotate any declaration viz.. class, interface, methods, constructors, fields, parameters and enum constants and other annotations.
- All annotations automatically extend the interface Annotation belonging to package java.lang.annotation.

contd..

- Retention Policy – determines at what point an annotation is discarded.
  - three retention policies. – SOURCE, CLASS and RUNTIME. RUNTIME retention offers highest persistence.
  - the default policy of **CLASS** is used.
  - A retention policy is specified for an annotation by using one of Java's built-in annotations.

contd..

- Using default values – Annotation members are given default values, which are used if no value is specified.
  - @ interface exampleanno {

    String str() default "Testing";

    int val() default 9000; }
  - @exampleanno() // both str and val default
  - @exampleanno(str = "some string") // val defaults
  - @exampleanno(val = 100) // str defaults
  - @exampleanno(str = "Testing", val = 100) // no defaults
- Marker Annotations – are empty annotations. Used in situations where only declarations are needed.
  - @Retention(RetentionPolicy.RUNTIME)
  - @interface sampleanno{ }

contd..

- Single member annotations – are the ones which contain only one member. The advantage being that when applied the name of the member need not be mentioned.
  - The name of the member must be **value**.
- Rules of annotations usage –
  - Annotations cannot inherit another annotation.
  - All methods declared inside annotation must be without parameters and the methods cannot be generic.
  - Annotation methods cannot specify a throws clause.
  - Annotation methods must return one of the following –
    - √ A primitive type, such as **int or double**
    - √ An object of type **String, Class, enum type**
    - √ Another annotation type
    - √ An array of one of the preceding types

Thank You