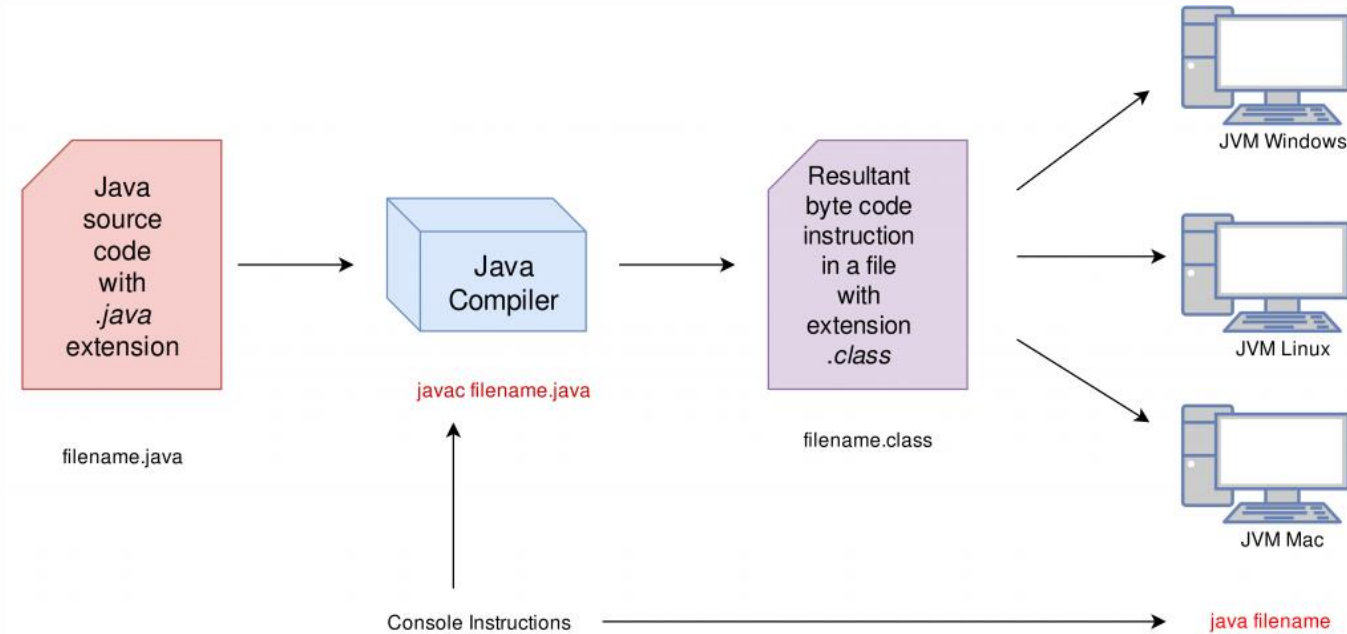# Core Java

Java Fundamentals

Introduction to Java

Brief History of Java

- Java was originally called "Oak"

- Java was meant to be a programming language specifically for appliances and various small devices

- The original specification:
  - Targeted set-top boxes for cable television that allowed users a control and interactivity when using the service
  - Java specification is owned by Sun Microsystems.

- The newly emerging Internet proved to be a much better platform for the specification:
  - The way the Net was being used had the same interactivity they planned for cable television customers
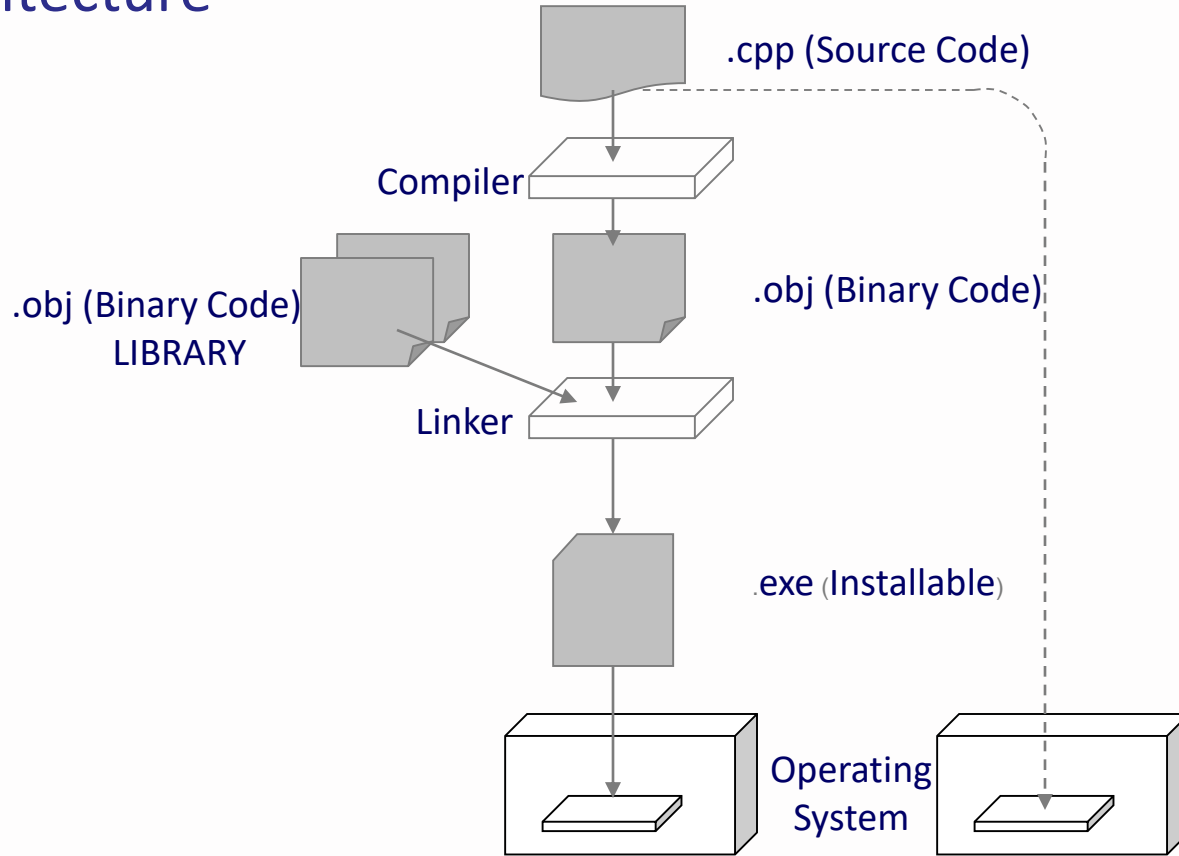
LTI
Let's Solve

The Java Technology Platform

- Java applications run on a virtual machine environment that:
  - Isolates the underlying platform
  - Achieves portability and performance
  - Provides security

- Java source code is written as plain text files (.java) that are compiled as platform independent byte-codes (.class)

- Java byte-codes are interpreted and executed by the virtual machine that passes the instructions to the actual platform
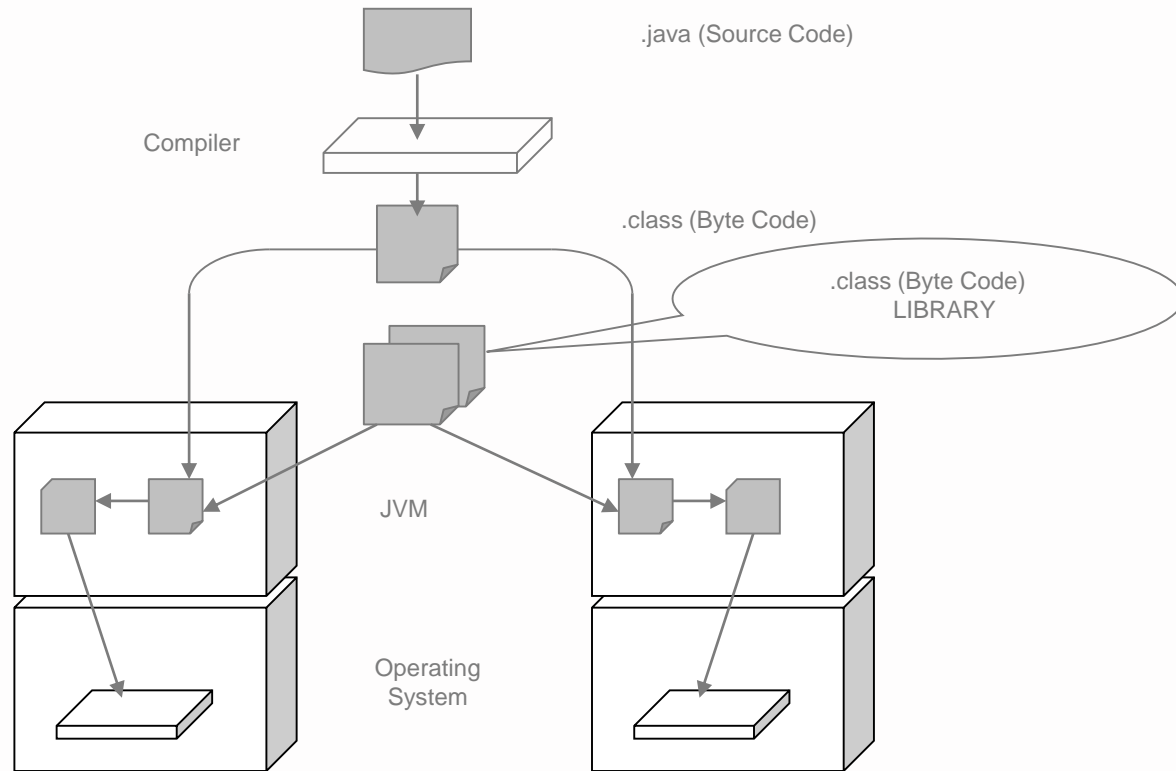
LTI

Let's Solve

The Java Technology Platform

C++ Architecture

.cpp (Source Code)

Compiler

.obj (Binary Code)
LIBRARY

.obj (Binary Code)

Linker

.exe (Installable)

Operating
System

Java Architecture

.java (Source Code)

Compiler

.class (Byte Code)

.class (Byte Code)
LIBRARY

JVM

Operating
System

Let's Solve
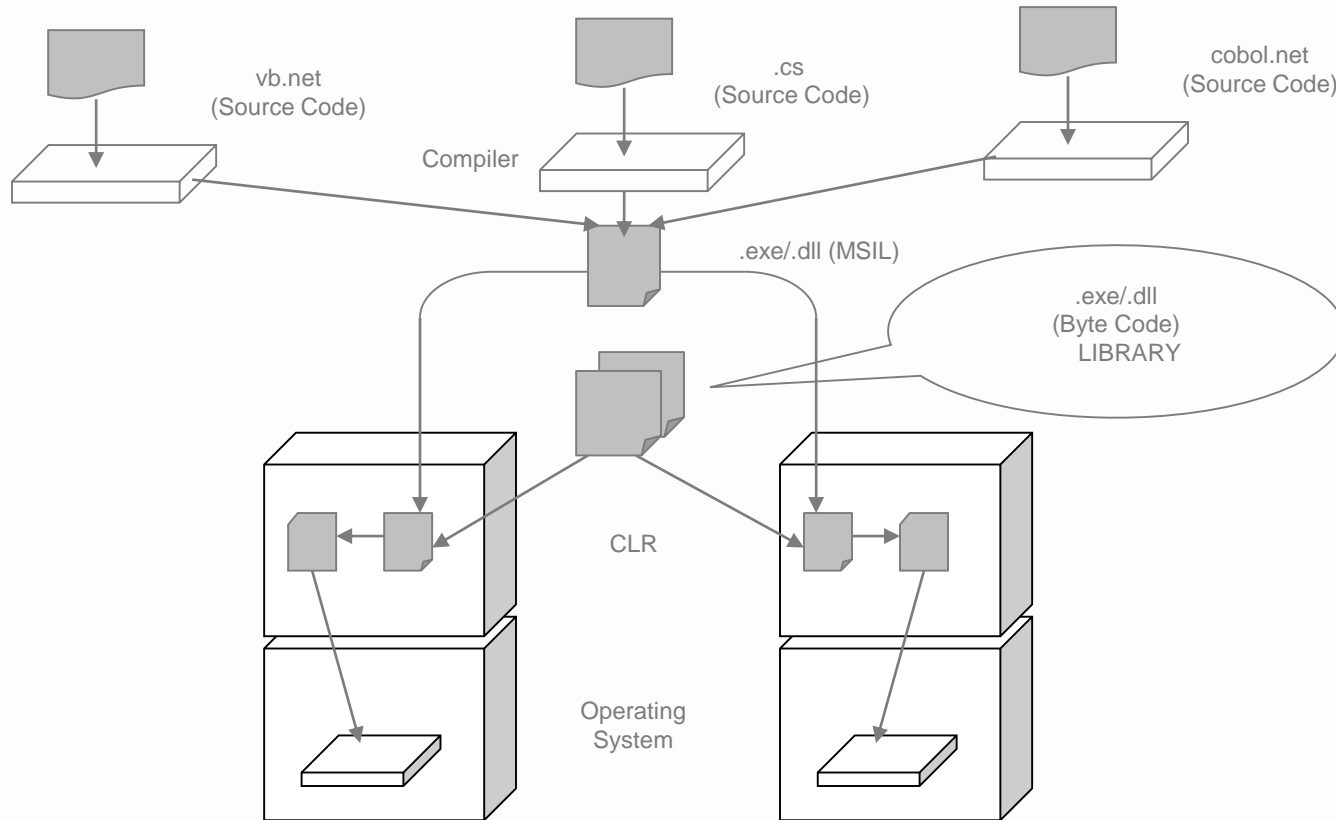
.NET Architecture

Java Development Kit

- Java Development Kit (JDK):
  - Is a set of software, tools and libraries that need to be installed in order to start writing and compiling Java applications
  - Installs the virtual machine (runtime environment) needed in order to execute Java applications on the platform
    - The virtual machine can be downloaded separately from the JDK as a JRE (Java Runtime Environment) download

LTI
Let's Solve

Integrated Development Environment

- Java source files are in text format and can be written using any text editor

- For greater productivity, an Integrated Development Environment (IDE) is highly recommended. An IDE:

  - Provides you with a set of tools that assist in developing, testing, and debugging Java applications

  - Popular IDEs are :

    - Eclipse

    - IntelliJ

    - NetBeans

LTI

Let's Solve

A simple Java program

//File: SampleApp.java

```java
package pack1;

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

}
```

Compile the java > javac SampleApp.java

Execute the .class file > java SampleApp

Java Fundamentals

Introduction to Java : Quiz

What does a Java program gets compiled into?

Assembly Code

Machine code

Bytecode

None

LTI
Let's Solve

Which method is called to begin the execution of a Java program?

main

run

start

None

Java Fundamentals

Language fundamentals

Java Keywords

- Java recognizes a set of keywords as part of the Java language

- Java keywords are used to support programming constructs, such as class declaration, variable declaration, and control flow

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

const	goto

Primitive Data Types

Type	Bits	Lowest Value	Highest Value
boolean	(n/a)	false	true
char	16	'\u0000' [0]	'\uffff' [2^{16}-1]
byte	8	-128 [-2^7]	+127 [2^7-1]
short	16	-32,768 [-2^{15}]	+32,767 [2^{15}-1]
int	32	-2,147,483,648 [-2^{31}]	+2,147,483,647 [2^{31}-1]
long	64	-9,223,372,036,854,775,808 [-2^{63}]	+9,223,372,036,854,775,807 [2^{63}-1]
float	32	±1.40129846432481707e-45	±3.40282346638528860e+38
double	64	±4.94065645841246544e-324	±1.79769313486231570e+308

LTI

Let's Solve

Data Types

▪ A primitive data type assigned a value:

int x = 42;

```
X
42
```

▪ A reference data type assigned a value:

Date today = new Date();

```
today
```
- - - - - - - - - - - - →
```
Day  = 25
Month = June
Year = 200
```

LTI
Let's Solve

Basic vs Reference Variables

```java
class BasicVsReference {
    public static void main(String[] args) {
        int x = 10;
        String str = "LTI";
    }
}
```
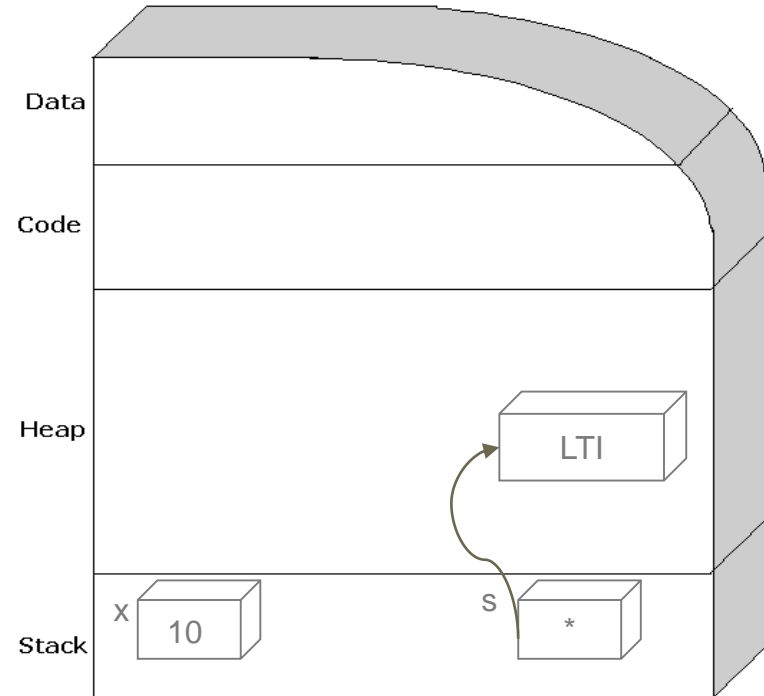
Parameter Passing

- Parameters in Java are Passed By Value

Passing Primitive Data Type Arguments

- A copy of the value is passed to the method.

- Any changes to the value exists only within the scope of the method.

- When the method returns, any changes to the value are lost. The original value remains.

Passing Reference Data Type Arguments

- A copy of the object's reference's value is being passed.

- The values of the object's fields can be changed inside the method, if they have the proper access level.

- When the method returns, the passed-in reference still references the same object as before. However, the changes in the object's fields will be retained.

LTI

Let's Solve

Types of Flow Control

- Sequential statements are executed in the order they are written

- This is the default flow of a program as instructions are executed in the order they appear
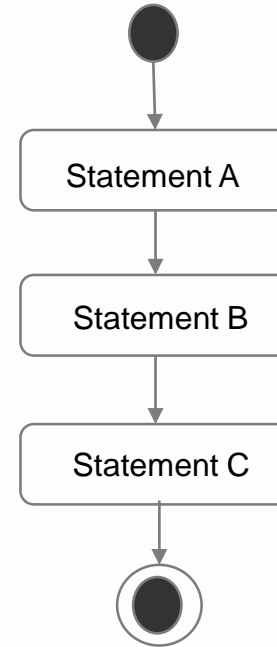
Types of Flow Control

- Selection structures execute a certain branch based on the results of a boolean condition

- This flow is useful in decision-making scenarios and can be implemented by using the if-else or switch-case statements
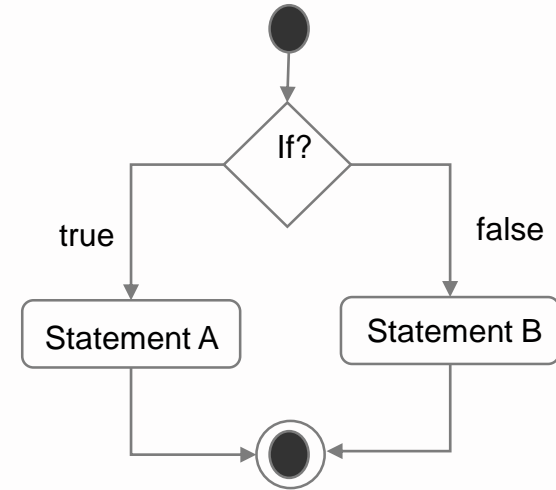
Types of Flow Control

- Iteration structures execute instructions repeatedly based on a condition

- This type of flow control can be implemented by making use of the different loop statements:
  - for-loop
  - while-loop
  - do-while loop
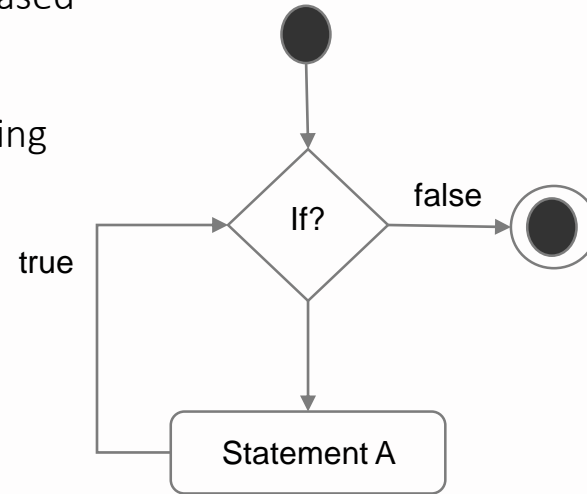
Java Fundamentals

Language fundamentals : Quiz

Java supports 50+ keywords which is/are the reserve words. Which of the following is Java keyword?

implements

default

outer

goto

Which of the following is NOT a Looping statement?

switch

for

while

do...while

LTI
Let's Solve

Java Fundamentals

Object Oriented Programming

Overview of Object-Oriented Programming

- Object-Oriented Programming is a programming pattern that makes use of objects and their interactions to design and implement applications

- Objects are entities that serve as the basic building blocks of an object-oriented application

- An object is a self-contained entity with attributes and behaviors

- In some way everything can be an object.

- In general, an **object** is a person, place, thing, event, or concept.

- Because different people have different perceptions of the same object, what an object is depends upon the point of view of the observer.
  - That is, we describe an object on the basis of the features and behaviors that are important or relevant to us.

Abstraction

- An object is thus an abstraction.

- An object is an "abstraction of something in the problem domain, reflecting the capabilities of the system to keep information about it, interact with it, or both." (Coad and Yourdon)

- An abstraction is a form of representation that includes only what is useful or interesting from a particular viewpoint.

  - e.g., a map is an abstract representation, since no map shows every detail of the territory it covers

Software Abstraction

- Abstraction is the process of reducing or simplifying information or a concept in order to retain that which is relevant to the current context

- Data is abstracted by high level language constructs, such as:
  - Numbers
  - Letters
  - Higher level data concepts

- Functionality is abstracted by presenting interfaces that hide complex logic or algorithms, such as:
  - Functions
  - Methods

- The mindset of an OO designer must be focused on creating small, simple re-usable components that can be combined to create more complex systems.

LTI

Let's Solve

Types of Objects

- Objects are usually classified as:
  - Objects representing physical things
    - e.g. students, furniture, buildings, classrooms

- Objects representing concepts
  - e.g., courses, departments, loan

LTI
Let's Solve

Class

- We classify similar objects as a type of thing in order to categorize them and make inferences about their attributes and behavior. This classification is generally defined as a Class

- A *class* is a template for a specific object

- A *class* defines the attributes and methods that all objects belonging to the class have

- The attributes and methods of a class are called 'fields' or 'members'

- An *instance* refers to an object that is a member of a particular class. All objects that belong to a class are instances of that class

Class and Objects

- A class is thus a type of thing, and all specific things that fit the general definition of the class are things that belong to the class
  - A class is a "blueprint" or description for the objects

- An object is a specific instance of a class
  - Objects are thus instantiated (created/defined) from

Ram   Gita   are instances of   **Student**
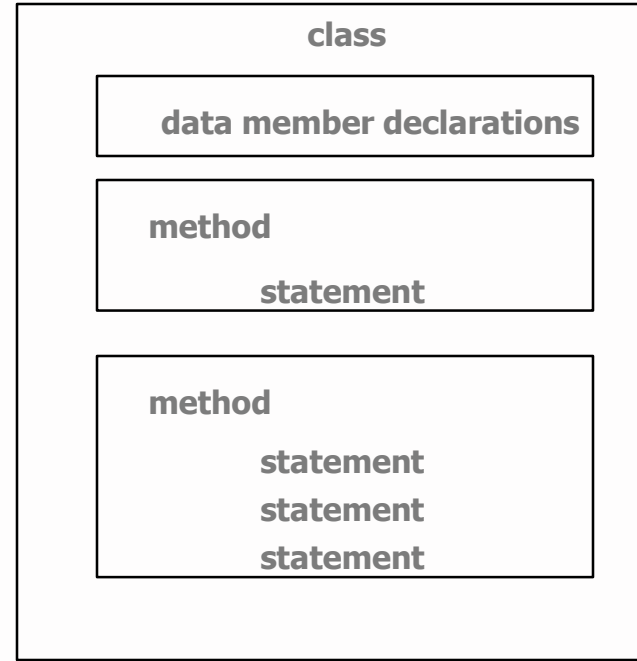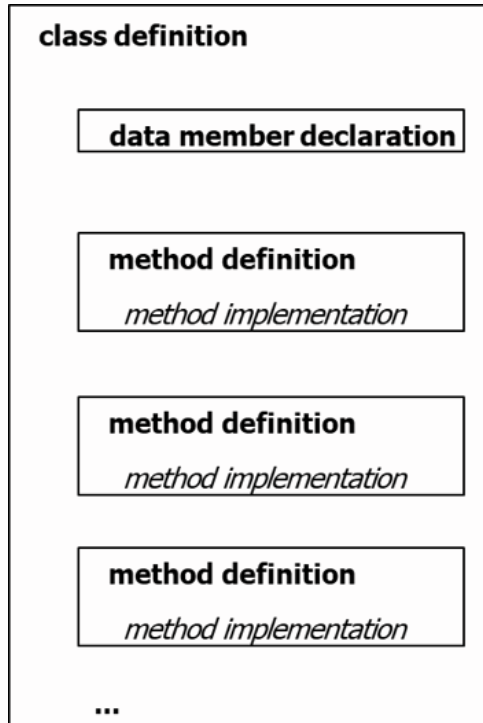
Seema   Ashok
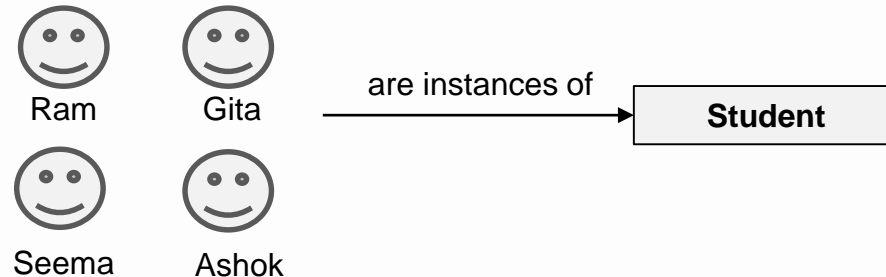
Interfaces

- The attributes and methods that a class provides to other classes constitute the class's interface

- The interface thus completely describes how users of that class interact with the class

Class Definition

- Classes are the most fundamental structural element in Java or C#
  - Every program has at least one class defined
  - Data member declarations appear within classes
  - Methods appear within classes
  - Statements appear within methods and properties

class
data member declarations
method   statement
method   statement   statement   statement

Let's Solve

Class Definition Example


```
class definition

    data member declaration


    method definition
        method implementation


    method definition
        method implementation


    method definition
        method implementation

    ...
```

```java
public class Rectangle{
    public double lowerLeftX,lowerLeftY;
    public double upperRightX,upperRightY;

    public double width() {
        return upperRightX - lowerLeftX;
    }

    public double height() {
        return upperRightY - lowerLeftY;
    }
    public double area() {
        return width() * height();
    }
}
```

Call or invoking the width and height methods defined above

Let's Solve

Types of members

- Two types of members (data members and methods):

  - **static** or **class** members

    - a member that is shared by all objects of that class

    - it is called a class member because it belongs to the class, not to a given object

    - it is called a static member because it is declared with the static keyword

  - **instance members**

    - Each instance/object of the class will have a copy of each instance member

Example


```
public class Person{

//class variables
static int numPersons;

//instance variables
String name;
String phone;
int age;

}
```

numPersons

Shared between all objects

Each object will get its own copy

name
phone
age

name
phone
age

Person objects created later by **new** statement

LTI
Let's Solve

Constructors

- When an object of a class is created (via new), a special method called a constructor is always invoked
  - You can supply your own constructor, or let the compiler supply one (that does nothing)
  - Constructors are often used to initialize the instance variables for that class

- Two characteristics:
  - constructor method has same name as the class
  - Constructor method never returns a value and must not have a return type specified (not even void)

LTI

Let's Solve

Brief History of Java

```java
public class Person{
String name;
String phone;
int age;

//class Constructor
public Person() {
age =-1;
}
//class methods
//instance methods
}
```

```java
Person boss = new Person();
```

```java
Person employee = new Person();
```

Each time a Person object is created a class constructor is called(and as a result employee.age and boss.age is initialized to -1

Encapsulation

- Perhaps the most important principle in OO is that of encapsulation (also known as information hiding or implementation hiding)
  - This is the principle of separating the implementation of an class from its interface and hiding the implementation from its clients
  - Thus, someone who uses a software object will have knowledge of what it can do, but will have no knowledge of how it does it
  - Benefits
    - Maximizes maintainability
      - making changes to the implementation of a well encapsulated class should have no impact on rest of system
    - Decouples content of information from its form of representation
      - thus the user of an object will not become tied to format of the information

LTI
Let's Solve

Encapsulation

- Encapsulation is implemented through access protection
  - Every class, data member and method in a Java or C# program is defined as either public, private, protected or unspecified/default

```
class ScopeClass{
    private int aPrivate = 1;
    int aUnknown = 2;
    public int aPublic = 3;
}
class ScopeTest{
    public static void main(String args[])
    {
        ScopeClass aScope = new ScopeClass();
         System.out.println("Scope Test");
         System.out.println("aPrivate = "+aScope.aPrivate);
         System.out.println("aUnknown ="+aScope.aUnknown);
         System.out.println("aPublic = "+aScope.aPublic);
    }
}
```

Generates a compile error

LTI
Let's Solve

Encapsulation

- An encapsulated object can be thought of as a black box or an abstraction

- Its inner workings are hidden to the client, which only invokes the interface methods
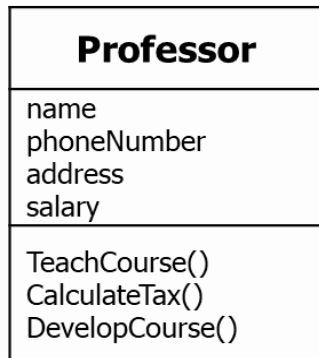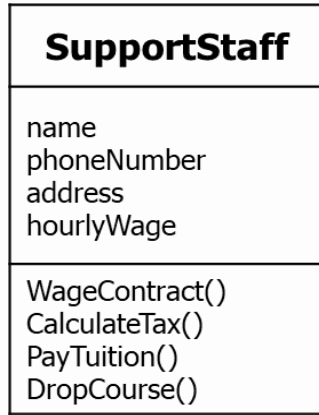
LTI
Let's Solve
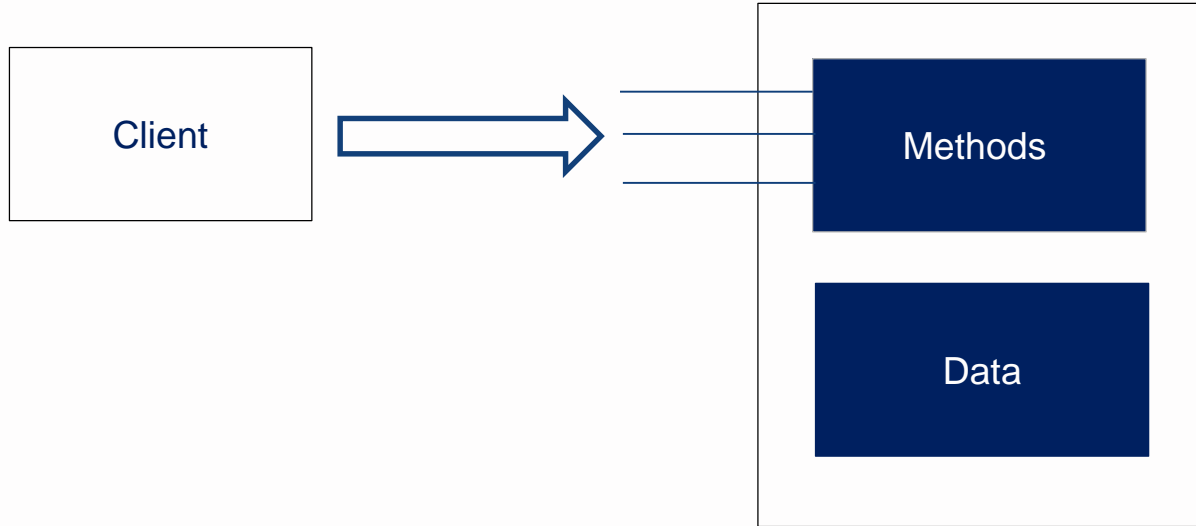
Class Relationships

- Classes do not exist by themselves themselves, but exist in relationships with other classes

- Kinds of relationship
  - Generalization (Inheritance)
  - Aggregation
  - Association
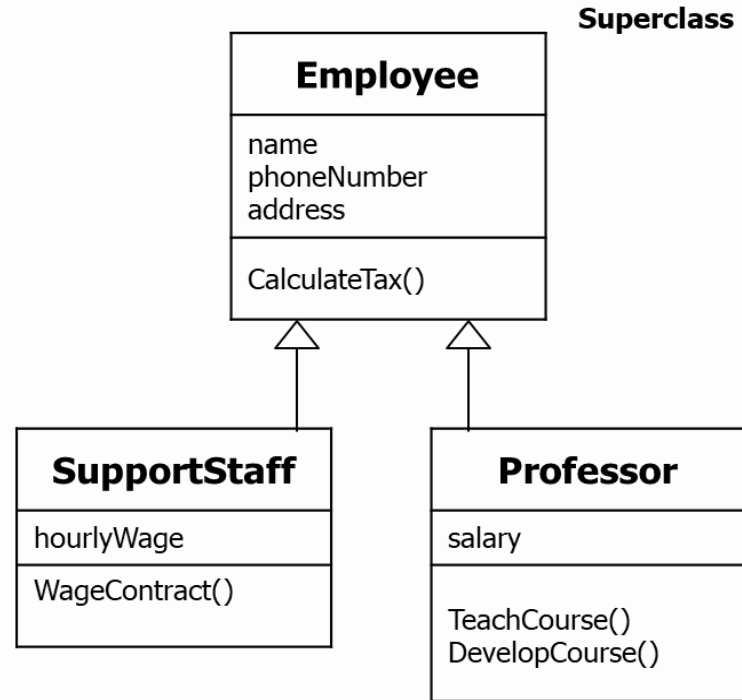 - Dependency

LTI

Let's Solve

Defining Inheritance

- **Inheritance** is the representation of an is a, is like, is kind of relationship between two classes
  - e.g., a Student is a Person, a Professor is kind of a Person
- With inheritance, you define a new class that encapsulates the similarities between two classes
- *A superclass*/base class/parent class – the class from which the attributes and behavior are derived
- *A subclass/*derived class/child class – a class that derives attributes and behavior from another class
- Inheritance is one of the language constructs that encourages the *re-use* of code by allowing the behavior of existing classes to be extended and specialized
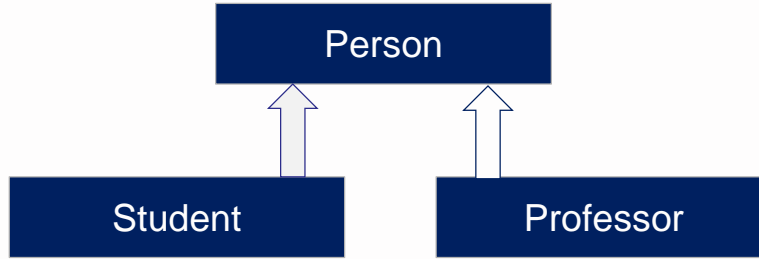
Let's Solve

Example

SupportStaff

name
phoneNumber
address
hourlyWage

WageContract()
CalculateTax()
PayTuition()
DropCourse()

Professor

name
phoneNumber
address
salary

TeachCourse()
CalculateTax()
DevelopCourse()

OR

Superclass

Employee

name
phoneNumber
address

CalculateTax()

SupportStaff

hourlyWage

WageContract()

Professor

salary

TeachCourse()
DevelopCourse()

LTI
Let's Solve

Brief History of Java

Person

Student   Professor
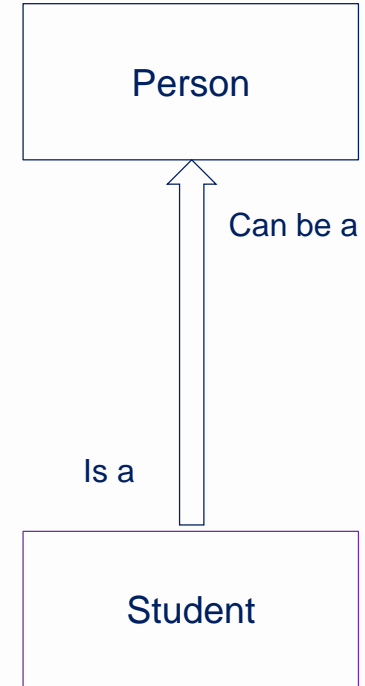
```
public class Person{
    //data methods and methods here
}

public class Student extends Person{

}

public class Professor extends Person{

}
```

Person

Can be a

Is a

Student

Let's Solve

Abstract and Concrete classes

- An *Abstract Class* is a class that provides common behavior across a set of subclasses, but is not itself designed to have instances of its own

- An abstract class is one that can not have objects instantiated from it

- An abstract class is designed as a template for other classes to follow by dictating behavior that must be implemented by its subclasses

- A concrete class is one that can have objects instantiated from it

- Defined using the 'abstract' class modifier

```
public class abstract Food {
        public abstract double calculateCalories();
                                }
```

LTI
Let's Solve

Polymorphism

- Polymorphism is:

- the ability to change the behavior of the application depending upon the type of an object

- the ability to manipulate objects of distinct classes using only the knowledge of their shared members

- It allows us to write several versions of a method in different classes of a subclass hierarchy and give them all the same name

- The subclass version of an attribute or method is said to override the version from the superclass

Let's Solve
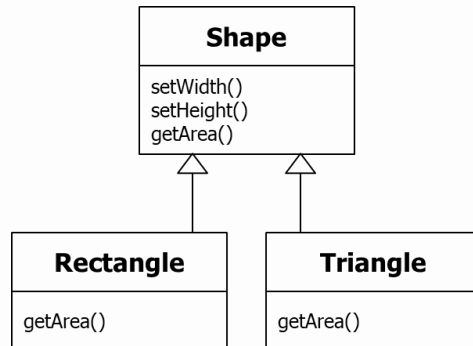
Example

```java
public class Shape{
    double width,height;
    public abstract double getArea()
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public void setHeight(double height) {
        this.height = height;
    }
}
```

```java
public class Rectangle extends Shape {
public double getArea() {
    return (width * height);
}
}
```
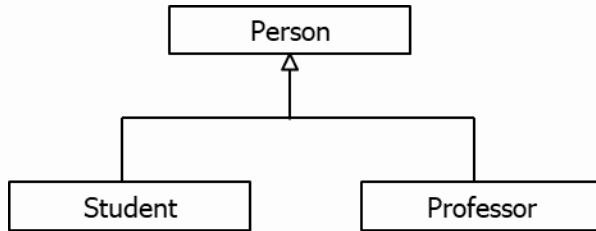
```java
public class Triangle extends Shape {
public double getArea() {
    return (0.5 * width * height);
}
}
```

```java
Rectangle myRect = new Rectangle();
myrect.setWidth();
myRect.setHeight();
System.out.println(myRect.getArea);
```

Shape
setWidth() setHeight() getArea()

Rectangle
getArea()

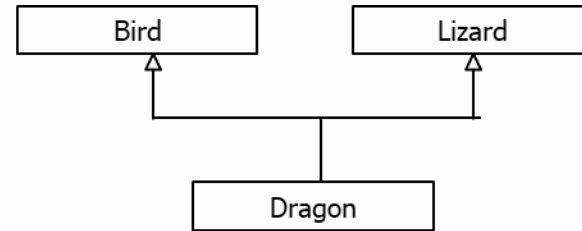| Triangle |
|----------|
| getArea() |

Let's Solve
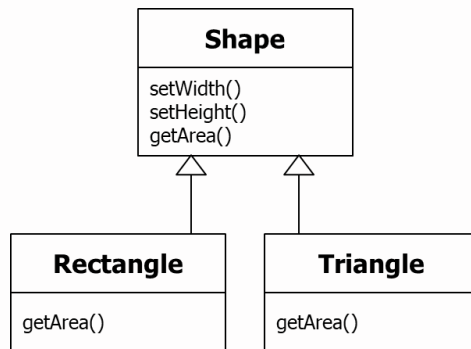
Single and multiple inheritance

▪ When a class inherits from only one other class, this is called single inheritance

▪ When a class inherits from two or more classes, this is called multiple inheritance

  ▪ Most OO languages do not support multiple inheritance (Java, C#, Smalltalk do not, but C++ does)

Single Inheritance

Multiple Inheritance

LTI
Let's Solve

Defining Java Interface

- An *Interface* defines a contract by specifying a set of method templates that an implementing class needs to follow

- An interface provides only a form for a class but no implementation

- An interface defines *what* a class can do but not *how* the class will do it

LTI
Let's Solve

Implementing Interfaces

- A class implementing interfaces is required to override the inherited methods

- Interfaces are implemented using the **`implements`** keyword

- Rules on implementing the interface methods
  1. Must have the same method signature and return type
  2. Cannot narrow the method accessibility
  3. Cannot specify broader checked exceptions

- Interface variables are implicitly **`public final static`**

- Interface methods are implicitly **`public abstract`**

Implementing Interfaces

- Example

```java
interface Animal {
    public void moo();
    public void sleep();
}


class Cow implements Animal {
    public void moo() {

        System.out.println("The cow says: moo");
    }
    public void sleep() {

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

LTI
Let's Solve

Substitutability

- Substitutability refers to the principle that subtypes must be substitutable for their supertypes
    - That is, if I write code assuming I have the superclass, it should work with any subclass
    - e.g., if I wrote code assuming I have a Person object, it should work even if I have a Student or Professor object
    - Substitutability is possible due to the polymorphic nature of inheritance
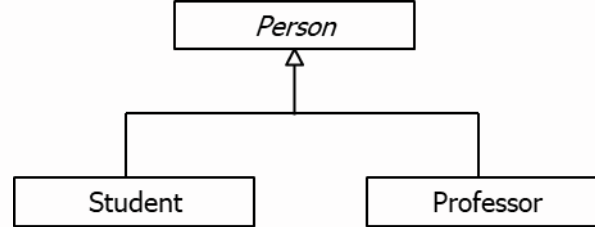
Reference Casting
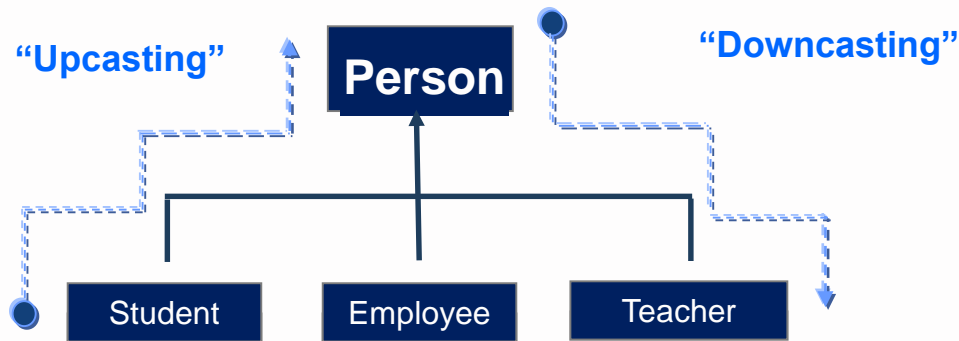
- *Upcasting* is conversion up the inheritance hierarchy

- To upcast a **Student** object, all you need to do is assign the object to a reference variable of type **Person.** Note that the **p** reference variable cannot access the members that are only available in **Student**

    * *Assuming Student is a subclass of Person*

    Person p = new Student(); *//upcasting*

LTI
Let's Solve

Reference Casting

- *Downcasting* is conversion down the inheritance hierarchy

- *To downcast* **Person** references an object of type **Student**, you can cast it back to **Student.** This time, it is called downcasting because you are casting an object to a class down the inheritance hierarchy

- Downcasting requires that you write the Student type in brackets
  - * *Assuming p is actually pointing to a Student object (as above)*

    Student s = (Student)p;
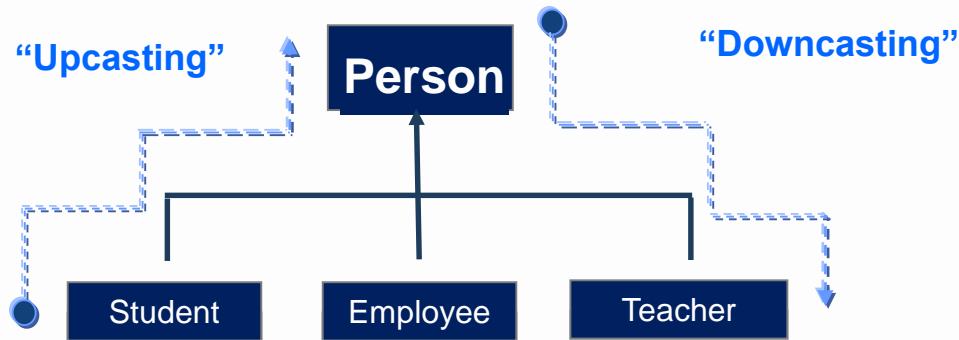
Virtual Methods

- A virtual method is a method whose actual implementation is dynamically determined during runtime

- All java methods(except static methods and final methods) are 'virtual' and can be overridden by methods and fields that belong to the sub class

- abstract methods can be considered as pure virtual methods in Java

Person
Introduce()

Student
Introduce()

Employee
Introduce()

Teacher
Introduce()
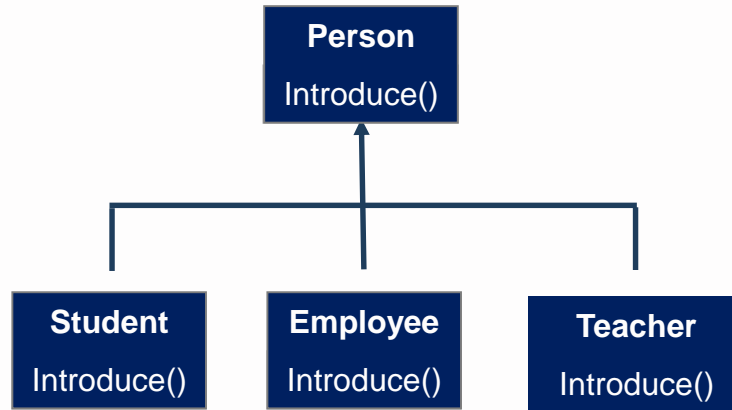
Brief History of Java

- A reference type variable can point to instances of its own type, or its subtypes through casting

  Person p = new Employee();

- When calling an object's methods through a reference variable, the implementation called is the one used by the object and not necessarily the reference variable

  p.introduce();

Access Modifiers Diagram

private

Private features of the Sample class can only be accessed from within the class itself.

Class

Package

default

Only classes that are in the package may access default features of classes that are in the package

Sample

Class

Class

Class

protected

Classes that are in the package and all its subclasses may access protected features of the Sample class.

Class

public

All classes may access public features of the Sample class.

Class

LTI
Let's Solve

Garbage Collection

- The *new* keyword explicitly allocates resources for an object, but there is no explicit way of telling the system to 'de-allocate' an object

- Garbage collection is the process of automatically clearing up the memory being used by objects that will no longer be used by the program

- Invocation of the garbage collector is automatically determined by the Java Virtual Machine, depending on the need to do so

- An application can 'suggest' that the system initiate garbage collection using *System.gc() or Runtime.getRuntime().gc()*

LTI

Let's Solve

Garbage Collection

- Garbage collection is used by Java to:
  - Recover resources (particularly memory) that are much needed by an application
  - Avoid dangling pointers
  - Ensure program integrity

- Dangling pointers are the occurrence of having variables or references pointing to objects in memory that are not of the appropriate type or value

Garbage collection

- Objects are safe from garbage collection if there is at least one reference variable that refers to the object

Person him = new Person();

Pointer / Reference him → Person Object

- Objects are eligible for garbage collection when there are no more identifiers that refer to the object

him = null;

him → null   Person Object

You want a class to have access to members of another class in the same package. Which is the most restrictive access that accomplishes this objective?

public

private

protected

default

Java does support:

Multiple inheritance

Multilevel inheritance

Java Fundamentals

Strings

Strings

▪ String Pool is a storage area in Java heap

▪ Strings are immutable.

▪ Each time a string literal is created, the JVM checks the string literal pool first. If the string already exists in the string pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new String object initializes and is placed in the pool.

To create a String object in Java, there are two ways:

▪ Using the new operator. For example,
  ▪ String s1 = new String("Ram");

▪ Using a string literal or constant expression. For example,
  ▪ String s1="Ram"; (string literal) or
  ▪ String s1="Ra" + "m"; (string constant expression)

Let's Solve

String basic methods

Method	Description	Return type
charAt(int index)	returns char value for the particular index	char
length()	returns string length	int
substring(int beginIndex)	returns substring for given begin index.	String
substring(int beginIndex,int endIndex)	returns substring for given begin index and end index.	String
contains(char c)	returns true or false after matching the sequence of char value.	boolean
concat(String str)	concatenates the specified string.	String
toLowerCase()	returns a string in lowercase	String
toUpperCase()	returns a string in uppercase	String

StringBuffer and StringBuilder

- The **StringBuffer** and **StringBuilder** classes are used when there is a necessity to make a lot of modifications to Strings of characters.

- Unlike Strings, objects of type StringBuffer and String builder can be modified over and over again without leaving behind a lot of new unused objects i.e they are mutable.

- StringBuffer is thread-safe and synchronized whereas StringBuilder is not. That's why StringBuilder is faster than StringBuffer.

- For String manipulations in a non-multi threaded environment, we should use StringBuilder else use StringBuffer class

Java Fundamentals

Strings : Quiz

Which of the following statements are true about String in java?

We can extend String class

String class is defined in java.util package.

String is immutable

String is thread-safe

String is case sensitive

Introduction

- If the requirement is:
  - *Open a file*
  - *Read a line from the file*

- *What we will have to end up doing is:*
  - *Open a file*
  - *If the file doesn't exist, inform the user*
  - *If you don't have permission to use the file, inform the user*
  - *If the file isn't a text file, inform the user*
  - *Read a line from the file*
  - *If you couldn't read a line, inform the user*
  - *etc. etc..*

Let's Solve

Introduction

- When developing applications, errors can occur due to multiple reasons:
  - Design errors or coding errors
  - Errors caused by the user of the application
  - External factors like disk, network, database, server, … failure, etc…

- It is our responsibility to produce quality code that does not fail unexpectedly
  - Consequently, we must design error handling into our programs.

- In Java, **exception handling** is the general term used for handling errors which occur during the execution of the program

- There are four important predefined classes in that part of the class hierarchy:
  - Throwable
  - Error
 - Exception & RuntimeException

LTI
Let's Solve

Exception Handling

- Exceptions fall into two categories:
  - Checked Exceptions
  - Unchecked Exceptions
- Checked exceptions
  - Checked exceptions are inherited from the core Java class Exception. They represent exceptions that are frequently considered "non fatal" to program execution
  - Checked exceptions must be handled in the code by default.
- Unchecked exceptions
  - Unchecked exceptions represent error conditions that are considered to be infrequent in the execution of a program
  - Unchecked exceptions need not be compulsorily be handled in the code

Errors

- Errors represent critical errors that should not occur and that the application is not expected to recover from

- Errors are typically generated from mistakes in program logic or design and should be handled through correction of design or code

- Errors are unchecked in nature similar to RuntimeException

- Examples: OutOfMemoryError, StackOverFlowError

Exception Class Hierarchy
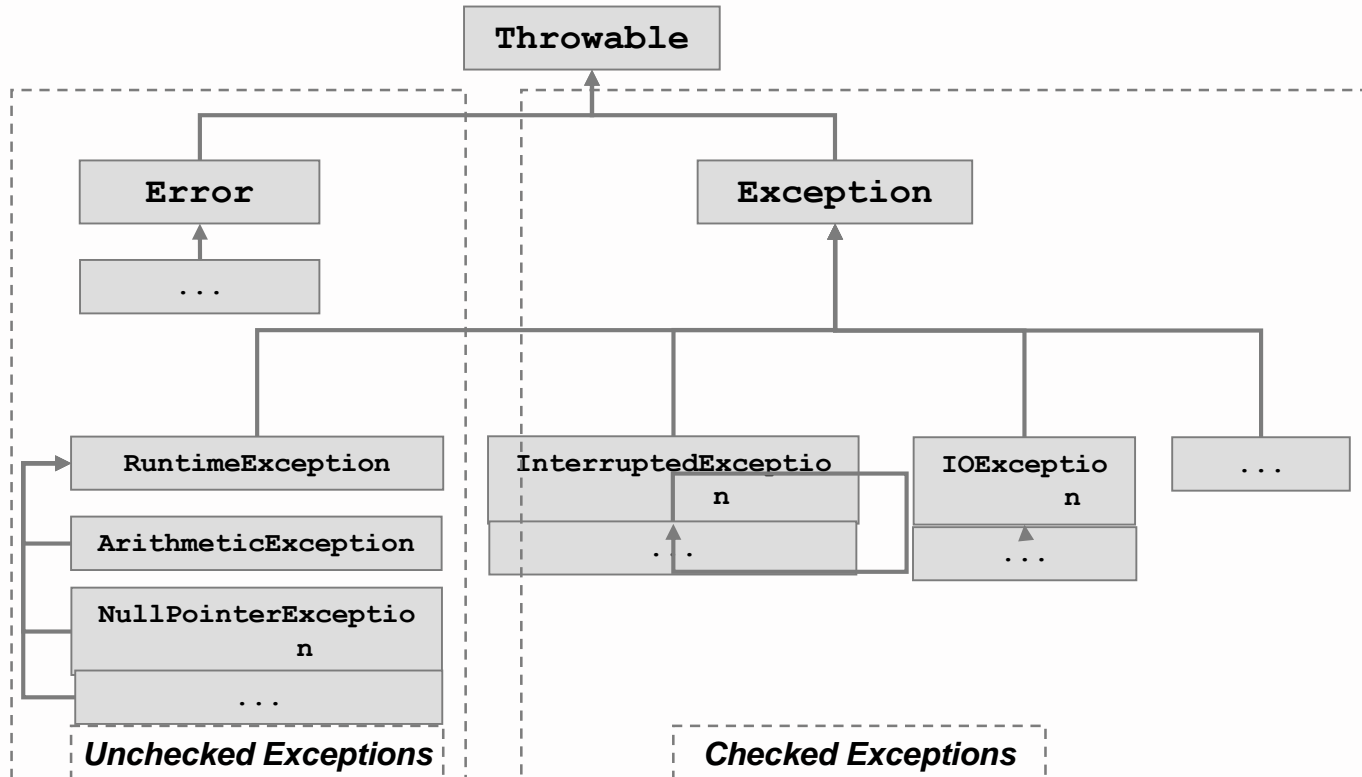
Handling Exceptions

- Exception handling mechanism is built around the *throw-and-catch* paradigm:
  - "*to throw*" means an exception has occurred
  - "*to catch*" means to deal with or handle an exception

- If an exception is not caught, it is *propagated* to the call stack until a handler is found

- Statements or methods called that declare that they might throw a checked Exception is required to be handled

LTI
Let's Solve

Using try-catch-finally Blocks

```
try {
    /*
     * some codes to test here
     */
} catch (SQLException sx) {
    /*
     * handle Exception1 here
     */
} catch (IOException ix) {
    /*
     * handle Exception2 here
     */
} catch (Exception ex) {
    /*
     * handle Exception3 here
     */
} finally {
    /*
     * always execute codes here
     */
}
```

`try` block encloses the context where a possible exception can be thrown

each `catch()` block is an exception handler and can appear several times

An optional `finally` block is always executed before exiting the `try` statement.

LTI
Let's Solve

Java Fundamentals

Exception Handling :Quiz

Which class is the parent of all the checked exceptions in Java?

ArithmeticException

RuntimeException

NullPointerException

Exception

Let's Solve

Which block is always executed whether an exception occurs or not?

throw

catch

finally

throws

Let's Solve

Java Fundamentals

Core Java API : Collections

Overview of the Java Collections API

- The Java Collections API is a set of interfaces and implementations included in the Java standard library

- A Collection or a Container is an object that groups together multiple elements into a single unit

- Each collection implements different storage, retrieval and manipulate behaviors

- The API contains *Interfaces* that specifies the abstract behavior of each collection

- The API contains concrete *implementations* of the interfaces

- Each concrete implementation uses different algorithms and behavior

Overview of the Java Collections API (cont.)
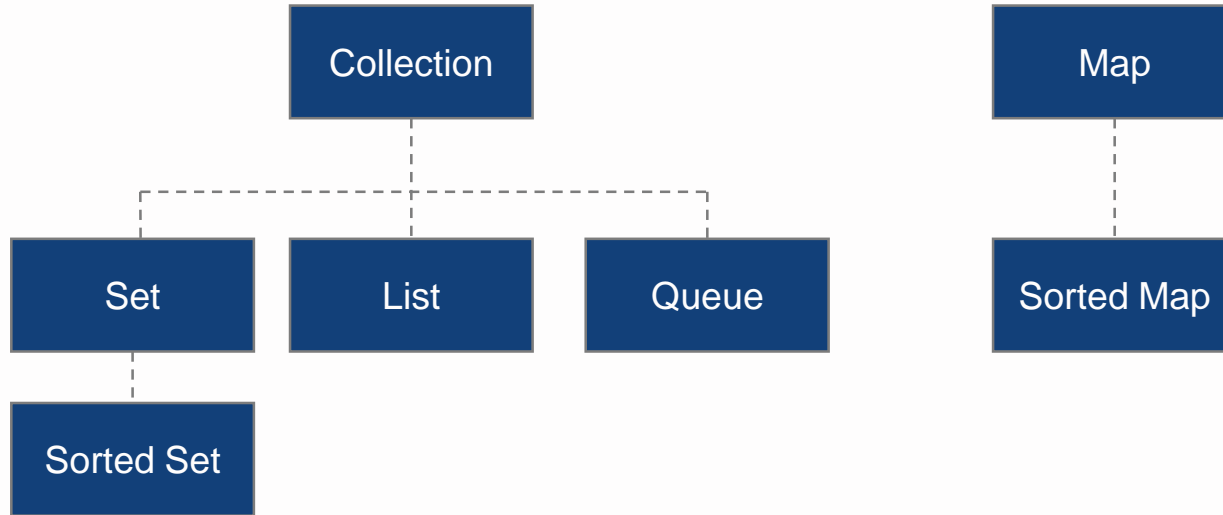
Collection Interface

- Root of the collections hierarchy.

- The *Collection* interface represents general purpose behavior for the interfaces in the hierarchy

- Does not have a concrete implementation

- Used in order to obtain a general reference to any kind of Collection

Set Interface

- A Set is a Collection that does not allow duplicate entries

- A Set Has the same methods as the Collection interface

- Can contain at least 1 null element

- Commonly used implementations are HashSet, LinkedHashSet and TreeSet

- Elements in a HashSet and LinkedHashSet are not sorted

- Elements in a TreeSet are sorted

LTI
Let's Solve

List Interface

- A List is an ordered collection of elements

- May contain duplicates

- An element's indexing is similar to arrays, with the first element in index 0

- Contains methods that allow manipulation of elements based on it's position in the sequence

- Common implementations are ArrayList and LinkedList

Let's Solve

Queue Interface

- A Queue is a collection for holding elements typically in a First-In-First-Out order

- Has versions for a set of methods inherited from the Collections interface that return special values instead returning an exception when failing

	Throws Exception	Returns special value
Insert	add()	offer()
Remove	remove()	poll()
Examine	element()	peek()

LTI
Let's Solve

Map Interface

- A Map is a collection that pairs a *key* to an element contained in the collection

- The key and the element can be any Object

- A key is assigned to an element when it is added to the map using the put(<key>, <element>)

- The key is used to retrieve an element from the Map using the get(<key>) method

LTI
Let's Solve

Ordering

- A collection can be sorted using the **_Collections.sort()_** method

- Elements in a collection implement the _Comparable_ interface which defines the sorting order

- Elements that do not implement the _Comparable_ interface cannot be sorted and will throw an exception

LTI
Let's Solve

Comparable Interface

- The *Comparable* Interface can be implemented by any class that wants to define its *natural ordering*

- Classes that implement the Comparable interface will need to define the ***compareTo(Object)*** method which defines how an instance of the class orders against the specified object

- The ***compareTo(Object)*** method should return:
  - 0 if the instance and the specified object are equal
  - A negative integer if the instance is 'less' than the specified object
  - A positive integer if the instance is 'greater' than the specified object

Comparator Interface

- The *Comparator* interface is similar to the Comparable interface except that it accepts two objects to be compared as parameters

- The *Comparator* interface can be implemented to compare two objects that are not *Comparable*

- Classes implementing the *Comparator* interface will need to override the **compare(Object A, Object B)** method to return the following:

  - 0 if A and B are equal

  - A negative number if A is 'less' than B

  - A positive number if A is 'greater' than B

LTI
Let's Solve

Java Fundamentals

Core Java API : Collections : Quiz

Which collection can be used for storing data in the same order as it was inserted?

HashMap

HashSet

ArrayList

A HashMap allows the existence of:

Null values

One null key

None from the options

Both null values and one null key
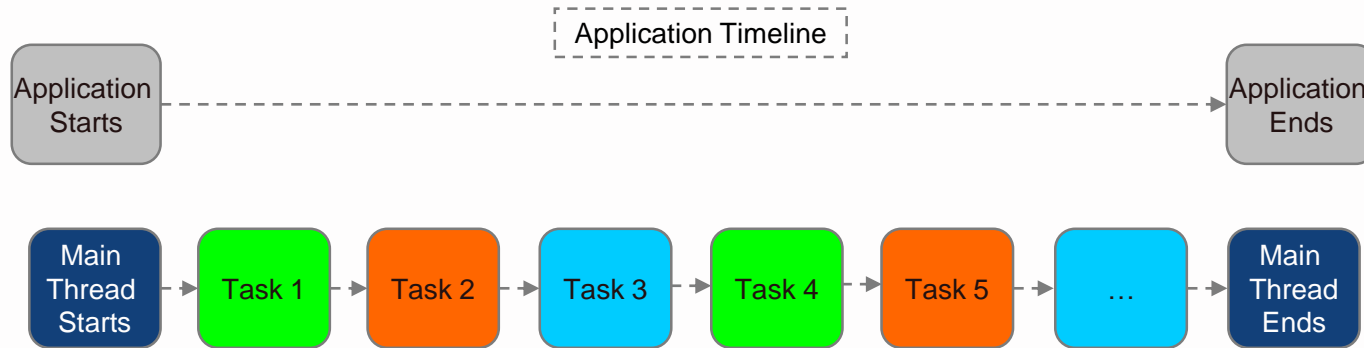
Let's Solve

Java Fundamentals

Concurrency

What is a Thread?

- The term *thread* is short for *thread of execution*

- A thread of execution is a sequence of instructions that is executed one after another in its own call stack

- A thread executes instructions one at a time; the thread must execute and finish the current instruction before moving on to the next one

- From a developer's perspective an application begins with 1 main thread that is usually started when the *public static void main(String[] args)* method is called

Single Thread Application (Diagram)

- Statements are executed one at a time in the appropriate sequence depending on program flow

- One statement must finish execution before the next statement can begin execution

Application Timeline

Application Starts — → Application Ends

Main Thread Starts → Task 1 → Task 2 → Task 3 → Task 4 → Task 5 → … → Main Thread Ends

Concurrency

- An application that is able to manage and coordinate multiple tasks simultaneously or pseudo-simultaneously is called a *concurrent application.*

- Concurrency allows tasks to execute without waiting for the other tasks to complete through a defined system of task switching and scheduling
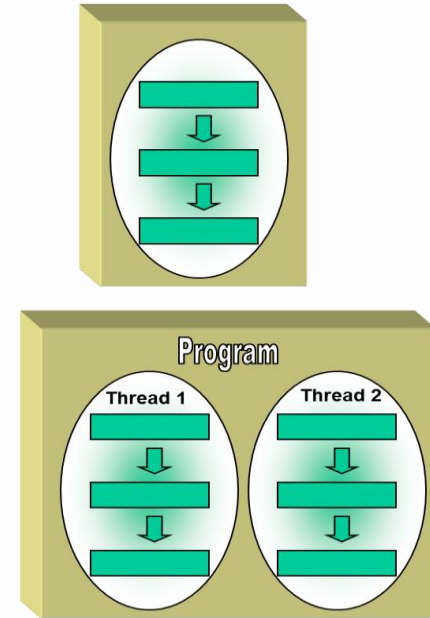
Concurrency in Java

- The most common way of implementing concurrency in a Java application is through multiple threads

- Java by nature is multithreaded as all applications have several background threads running in addition to the application code

- Each thread can create and start new threads

- The application terminates when the main thread and all application created threads finish execution

- Multiple threaded applications make use of thread switching and scheduling that allow multiple threads to make use of the system's resources

- The actual algorithm for thread scheduling depends heavily on the native operating system

LTI
Let's Solve

Creating a Thread

- An instance of the Thread class is needed to create and execute the new thread

- The code that the thread will execute can be defined through either of the following:
  - Subclass the Thread class, override its public void run() method
  - Pass a *Runnable* object with an overridden public void run() method to a Thread object's constructor

- Call the Thread instance's *start()* method to start the thread.

Thread Lifecycle

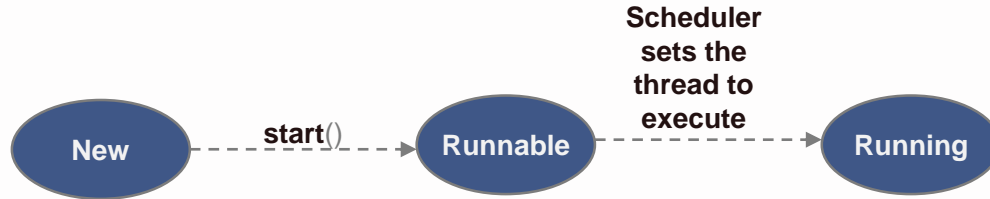- When a thread is created (but not started) it is at create state

New

- When a thread invokes its start() method, it is now *alive* and is placed in a *runnable pool* where all living threads wait their turn to be run by the scheduler
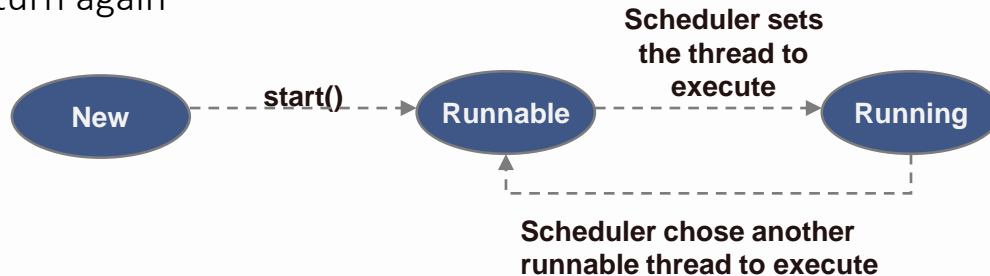
New — **start()** → **Runnable**

Thread Lifecycle (cont.)

- The thread scheduler chooses a thread to execute from the threads in the pool


- If another thread is chosen by the scheduler to execute before it finishes, it goes back to the thread pool to await its turn again
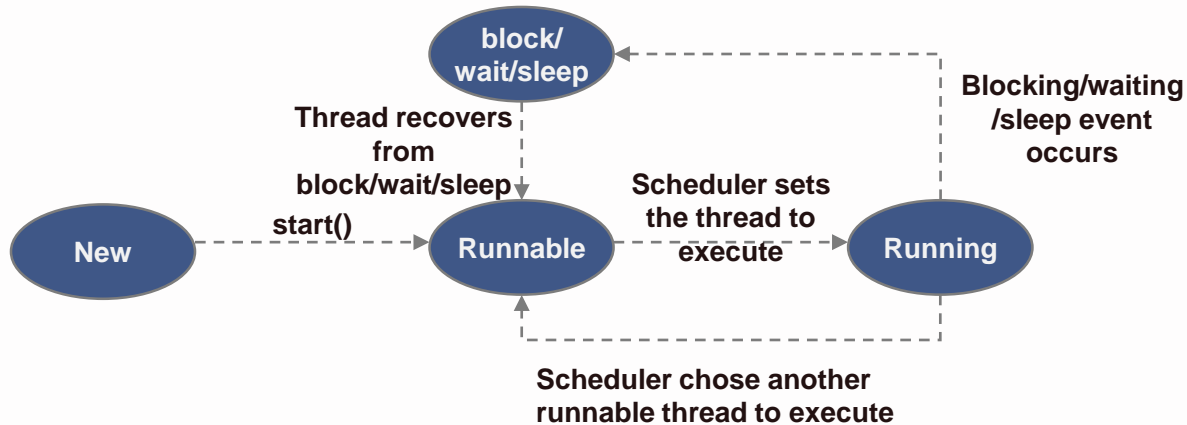
Thread Lifecycle (cont.)

- A thread may also block, wait or sleep while it was running causing it to enter a state where it is alive, but not runnable
  - blocking: thread is waiting for a resource or an event
  - waiting: thread is waiting for a signal from another thread
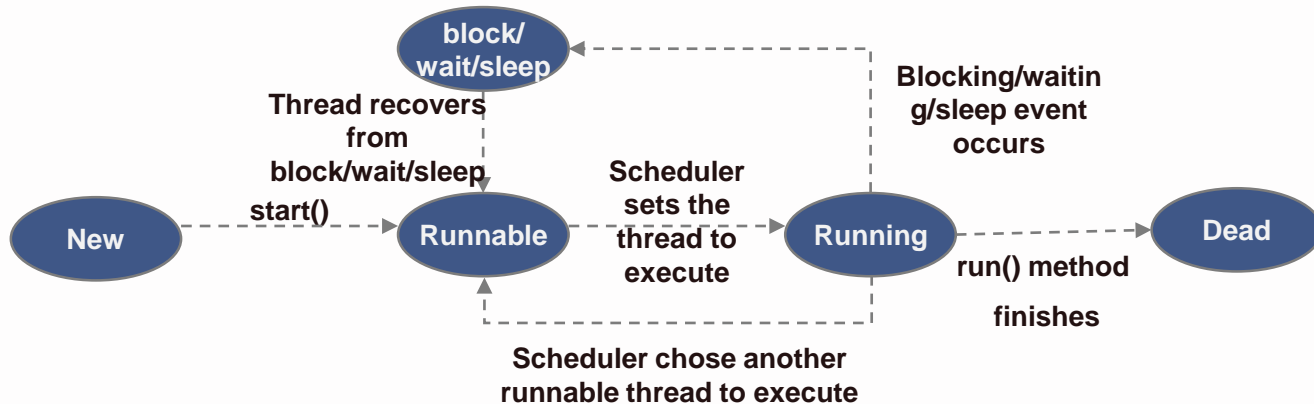 - sleep: the thread pauses itself for a specified duration
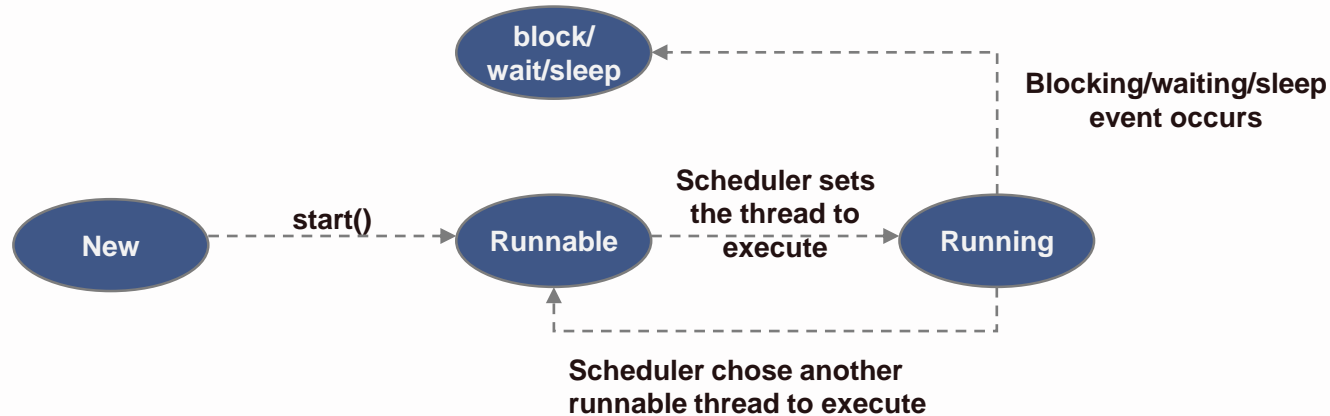
Thread Lifecycle (cont.)

▪ After a thread recovers from blocking, waiting or sleeping, it returns to a *Runnable* state and waits in the pool to be executed again

Thread Lifecycle (cont.)

▪ If the thread finishes its execution by exiting its run() method, it is now a 'dead' thread

Managing Threads

- Basic Concepts

- Setting Thread Priority

- Pausing Threads

- Yielding a Thread

- Joining a Thread

- Synchronizing Threads

LTI

Let's Solve

Basic Concepts

- The application often has very little control in determining the behavior of threads

- The application can influence the behavior of threads, but it cannot have direct control

- A variety of factors are taken in by the thread scheduler in determining which threads to execute, such as:
  - Availability of a CPU resources
  - Activity or inactivity of the thread
  - Designated priority

- Application logic should never be designed to rely on predicting thread behavior

- After a thread is caused to block, wait or sleep, it returns to the *runnable* state not the *running* state, making it impossible to determine exactly when it will run again

Thread Priority

- Normally a thread with higher priority will be scheduled to run before a thread with a lower priority, but this is not always the case

- The scheduler may choose to run a lower priority thread for efficiency reasons or if the thread itself causes the scheduler to choose another thread

- Implementation of thread priority varies depending on the underlying platform, and thus should never be considered as a factor when designing application logic

LTI
Let's Solve

Setting Thread Priority

- A Java thread inherits its default priority from the thread that created it

- A thread can set its priority by calling its *setPriority(int)* method

- Valid values are defined as integers ranging between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY

LTI
Let's Solve

Pausing a Thread

- Calling *Thread.sleep(duration)* stops the current thread from executing and places it in a state that prevents it from being chosen by the scheduler

- Putting a thread to sleep guarantees that it will not run and will not be eligible run for the specified duration

- When a thread wakes up after being put to sleep, it will be placed in a *Runnable* state, and NOT the *Running* state

Yielding a Thread

- The *Thread.yield()* method is intended to cause the current thread to go back to a *runnable* state, ideally allowing the scheduler to choose another thread to execute

- Yielding a thread does NOT guarantee that another thread will be chosen to run; there is still a chance that the thread that yielded will be chosen again

LTI
Let's Solve

Joining a Thread

- A thread can be told to "join a thread at the end" by calling the join() method of another *live* thread object

- This guarantees that the *current* thread will enter a *blocked* state while the thread object it joined finishes

- After the joined thread finishes execution, the current thread returns to a *runnable* state

Thread Interference

- Threads communicate and pass data to each other by accessing the fields of objects that they share

- Because of this shared nature it is possible for threads to *interfere* with each other with regards to the objects that they share

- Thread interfering with one another with regards to shared objects can result in inconsistent views on the data that they share

LTI
Let's Solve

Synchronization and Locks

- Synchronization is one of the tools to avoid thread interference and memory consistency errors when multiple threads try to access a shared resource

- Shared Resources
  - If one thread tries to read the data and other thread tries to update the same data, it leads to inconsistent state. This can be prevented by synchronizing access to the data

- Sections of code can be synchronized so that only one thread at a time can execute that section of code

- Synchronization relies on the concept of a 'lock' that a Thread has to acquire before being allowed to execute a section of code

- When a thread invokes a synchronized method, it automatically acquires the Intrinsic Lock for that method's object and releases it when the method returns

Example of multiple threads sharing the same object

```java
class   InternetBankingSystem
{
    public static void main (String args[])
    {
            Account accountObject = new Account();
            MyThread t1 = new MtThread(accountObject);
            MyThread t2 = new MtThread(accountObject);
            MyThread t3 = new MtThread(accountObject);
            t1.start();
            t2.start();
            t2.start();
            // some more operations
    } //end main
} // end class MyThread
```

account
(shared object)
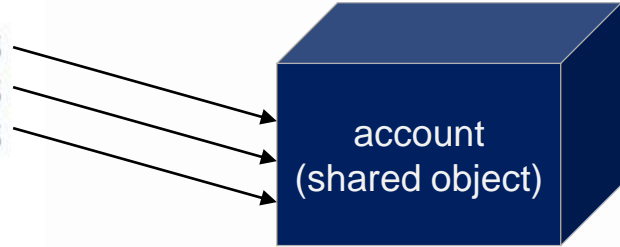
synchronized keyword

```
class  Account {
float balance;

// if 'synchronized' is removed, the outcome is unpredictable
public synchronized void deposit (float amt) {
balance +=amt;
}

public synchronized void withdraw (float amt) {
if (condition)
{
//some operations
balance -=amt;
}
}
}
```

Coordinating Threads

- Threads will often need to coordinate with one another to accomplish their tasks

- One of the more common coordination scenarios is a thread that needs to finish a section of code before another thread can proceed

- One way to accomplish this coordination is by making the dependent thread 'wait' until it has been 'notified' by the other thread that it can proceed

- Objects have a wait/notify mechanism that allows threads to coordinate with each other

- The wait() method of an object tells the current thread acting on the object to block

- The thread waiting on the object will continue once the object's notify() method has been executed by another thread

- The wait() and notify() methods of an object can only be executed by a thread that has exclusive lock on the object

LTI

Let's Solve

About Explicit Locks

- To address the limitations of Intrinsic Locks, Explicit locks from Java 5 onwards can be implemented to provide more flexibility

- Features of Explicit Locks:
  - Ability to interrupt a thread waiting for a specific lock
  - A thread can give a timeout value for attempting to acquire the lock
  - Read/write locks are supported– multiple threads can hold a lock simultaneously for read only access.
  - The traditional wait/notify metaphor is extended to allow conditions
  - Support for fairness by following the first-in-first-out order if more than one thread is waiting for a lock
  - The ability to lock beyond the scope of a block: for example, one method can pass a lock object to another thread
  - Ability to query the status and properties of a specific lock programmatically (i.e., number of threads waiting to acquire the lock)

Thread Synchronization revisited

- When multiple threads share an object and that object is modified by one or more of the threads, indeterminate results may occur

- The problem can be solved by giving one thread at a time exclusive access to code that manipulates the shared object. During that time, other threads desiring to manipulate the object are kept waiting. When the thread with exclusive access to the object finishes manipulating it, one of the threads that was kept waiting is allowed to proceed. In this fashion, each thread accessing the shared object excludes all other threads from doing so simultaneously. This is called **mutual exclusion**

- Java uses locks to perform synchronization. Any object can contain an object that implements the **Lock** interface (package java.util.concurrent.locks)

    - **ReentrantLock, ReentrantReadWriteLock** are the commonly used class from this package

- Also new techniques were introduced like **Semaphore, CountdownLatch, CyclicBarrier,** etc... provide new techniques for managing synchronization issues in our code

LTi
Let's Solve

ArrayBlockingQueue

- Java 5 includes a fully-implemented circular buffer class named **ArrayBlockingQueue** in package java.util.concurrent, which implements the **BlockingQueue** interface

- The BlockingQueue interface implements the Queue interface and declares methods **put** and **take**, the blocking equivalents of Queue methods offer and poll, respectively

- Method **put** will place an element at the end of the BlockingQueue, waiting if the queue is full

- Method **take** will remove an element from the head of the BlockingQueue, waiting if the queue is empty

- Class ArrayBlockingQueue implements the BlockingQueue interface using an array. This makes the data structure fixed size, meaning that it will not expand to accommodate extra elements

Java Fundamentals

Concurrency: Quiz

Maximum thread priority in Java is?

5

20

10

1

LTI

Let's Solve

A thread can acquire a lock by using which reserved keyword?

synchronized

locked

volatile

None from the options

Let's Solve

Java Fundamentals
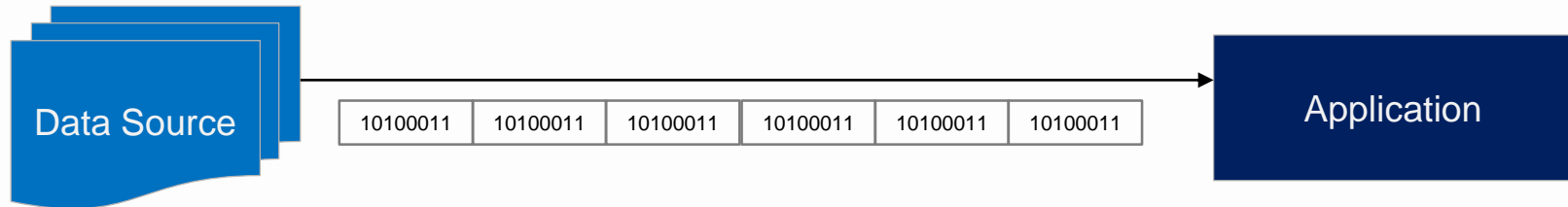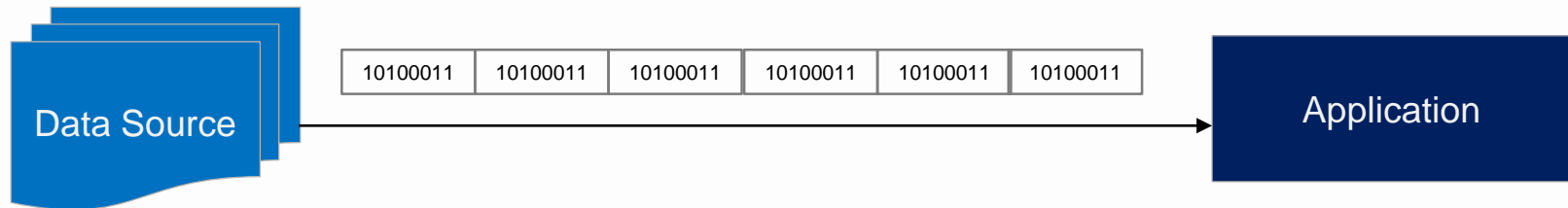
Core Java API: I/O

LTi

Let's Solve

I/O Streams

- The standard **java.io package**, is defined in terms of *streams*

- Streams are ordered sequences of data that have a
  - source (input stream), or
  - destination (output stream)

- Sending data from the application to a data destination is called *writing* to a stream

- Retrieving data from a data source is called *reading* from a stream

- Streams should be 'opened' for reading, then 'closed' after

Brief History of Java

- Reading from a data source (can be a file, a socket, a database, keyboard etc...)

| Data Source | 10100011 | 10100011 | 10100011 | 10100011 | 10100011 | 10100011 | → Application |

- Writing to a data destination (can be a file, a socket, a printer, a monitor etc.)

| Data Source | 10100011 | 10100011 | 10100011 | 10100011 | 10100011 | 10100011 | → Application |

Java I/O Package (cont.)

- The central classes in the package are Abstract **InputStream** and **OutputStream,** which are base classes for reading from and writing to byte streams, respectively

- Abstract **Reader** and **Writer** classes serve as base classes for various specialized stream subclasses that deals with characters

- The package also has a few miscellaneous classes to support interactions with the host **file system**

LTI
Let's Solve

Streams

Byte streams

InputStream   OutputStream

Character streams
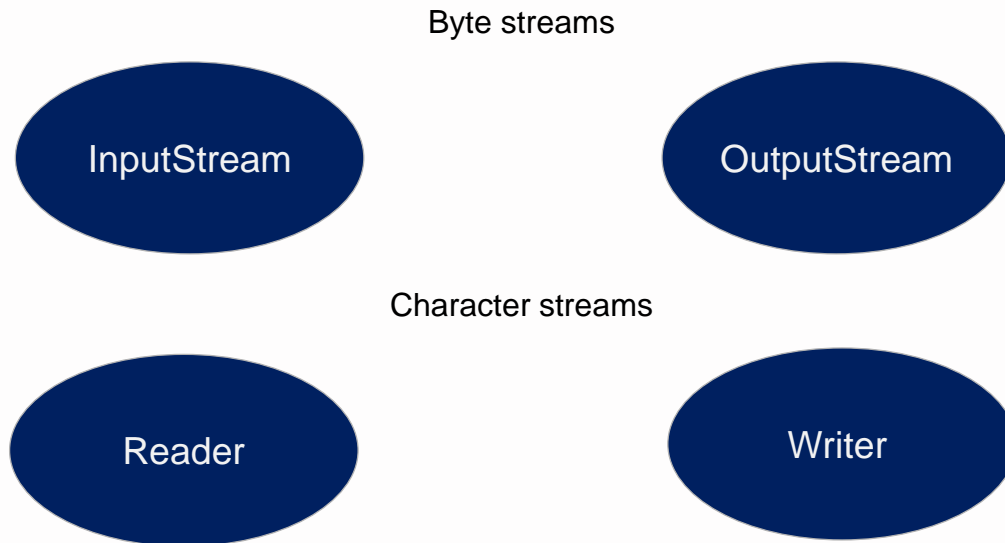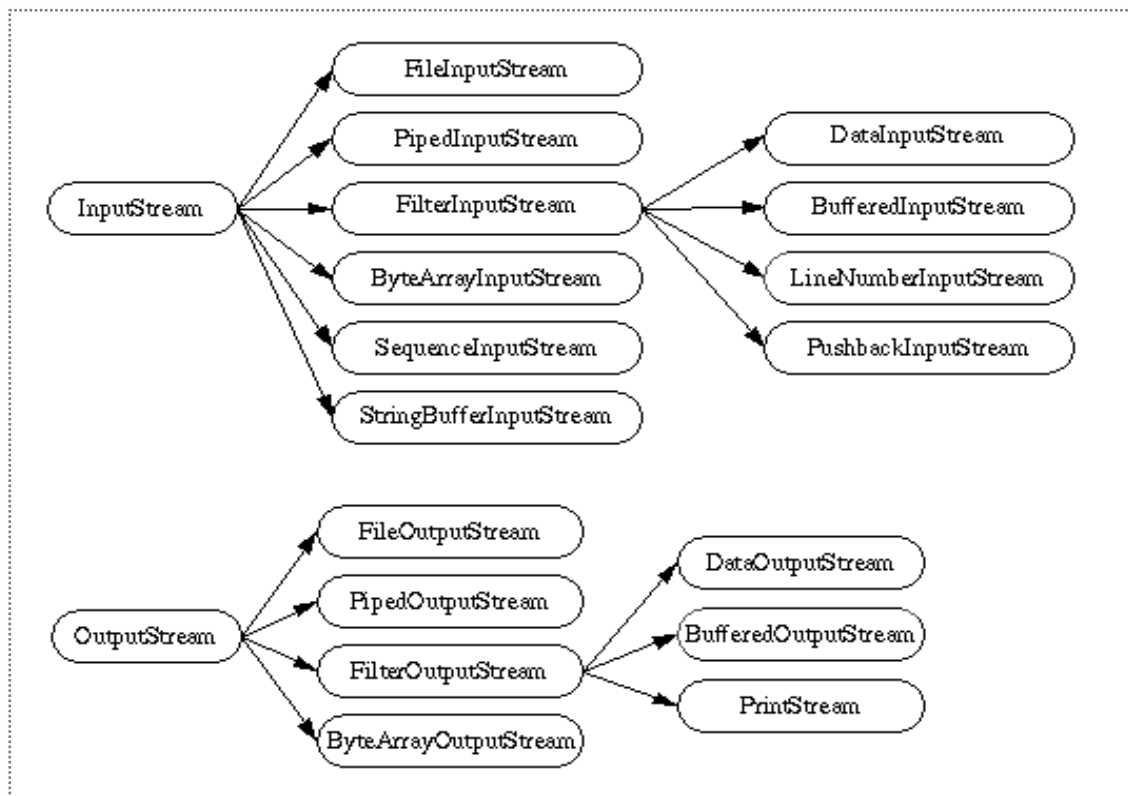
Reader   Writer

Streams

character streams

- Read and write 16-bit Unicode characters

- Reader and Writer

- Should be used for reading/writing text information

byte streams

- Read and write 8-bit (= 1 byte) characters

- InputStream and OutputStream

- Should be used for reading/writing images or sound files

- Used for Object serialization

Let's Solve

Java I/O Package (cont.)

Console IO

- These are the statements for displaying simple text messages to the user

System.out.print(...)

System.out.println(...)

- The **System.out** object is an instance of the **PrintStream** class, which is a type of output stream used for outputting characters

- The **System.in** object is an instance of an **InputStream** class, which allows reading of raw bytes from a data source

Reading from the Keyboard

- The **System.in** object is already created, open, and ready for use

- System.in is an instance of *InputStream* and will read keyboard entries as individual bytes

- Wrapping System.in with an instance of *InputStreamReader* allows us to read the entries as *characters*

- Wrapping the *InputStreamReader* with a *BufferedReader* to allow the reading and buffering of multiple characters and lines of text

LTI
Let's Solve

File IO

- Objects of type File are used to represent the actual files (but not the data in the files) or directories that exist on a computer's physical disk

    File thisFile = new File("sample.txt");

- *FileReader* and *FileWriter* classes can be used to wrap File objects for reading and writing

- BufferedWriter can be used to write lines of characters to a file

    Writer output = new BufferedWriter(new FileWriter("sample.txt"));

Serialization

- Serialization allows instances of an object to be represented as a stream, which can then be written to a data source/destination

- The interface *Serializable* needs to be implemented by any class whose instances are allowed to be serialized

- The field modifier 'transient' is used to mark class members that should NOT be serialized along with the other state attributes of the object

Let's Solve

Serialization

- Serialization provides a standard way to save and restore object graphs between sessions on the same or different systems.

- Serialization saves the following :

1. Object type

2. Internal information

3. References to other objects

LTI
Let's Solve
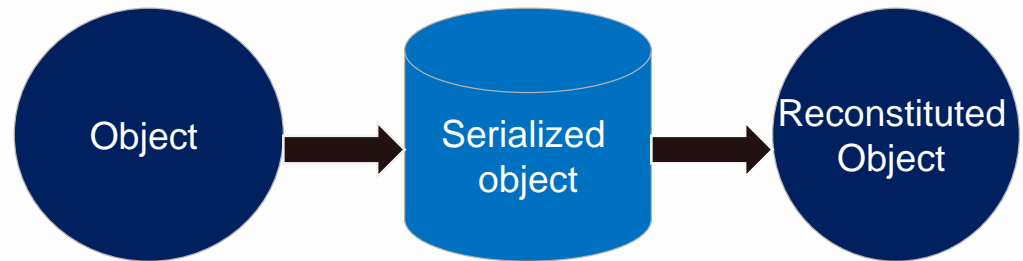
Marking an Object for Serialization
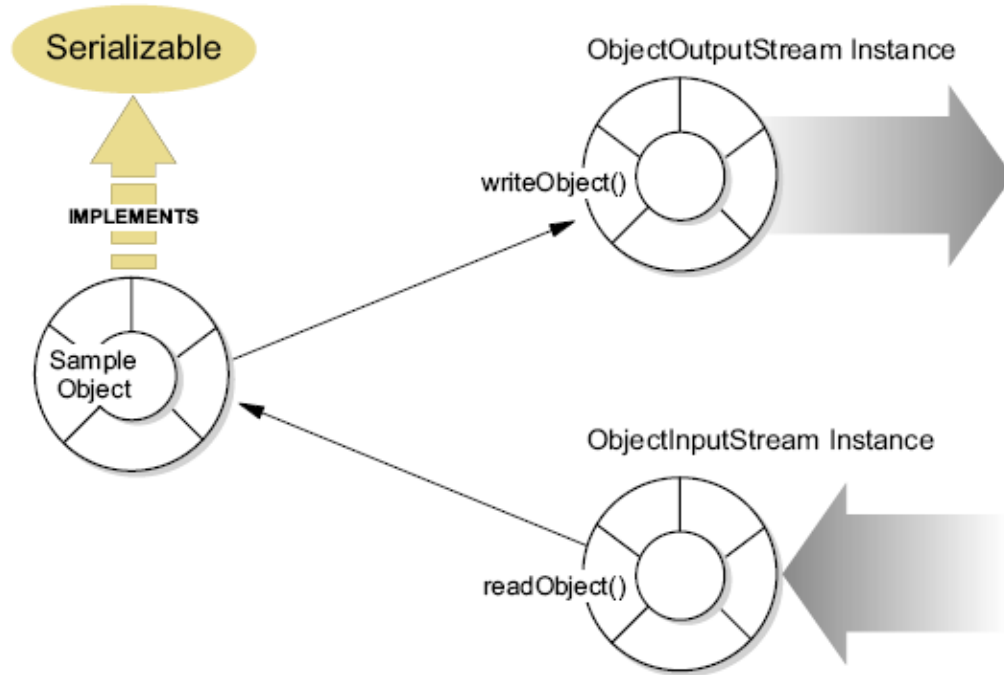
Serialization (cont.)

- To write an object to a stream, use an instance of the *ObjectOutputStream* class and chain it to an output stream destination

  ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(target));

  out.writeObject(aSerializableObject);

- To read a serialized object, use an instance of an *ObjectInputStream* class chained to an input stream from a data source.

  ObjectInputStream in = new ObjectInputStream(new FileInputStream(target));

  Object savedObject = in.readObject();

Java Fundamentals

Core Java API: I/O :Quiz

Which of these is used to perform all input & output operations in Java?

streams

variables

classes

methods

LTI
Let's Solve

# Java Fundamentals

Core Java API : JDBC

JDBC Overview

- JDBC (Java Database Connectivity) is the Java industry standard for database-independent connectivity between the Java language and other databases

- JDBC provides a comprehensive API that provides the application the ability to connect to databases, send SQL statements, and process results

LTI
Let's Solve

Database Drivers

- Database Drivers or JDBC Drivers are required to connect to different databases. The JDBC requires different drivers for each database

- JDBC drivers provide the connection to the database and implement the protocol necessary for sending queries and retrieving results
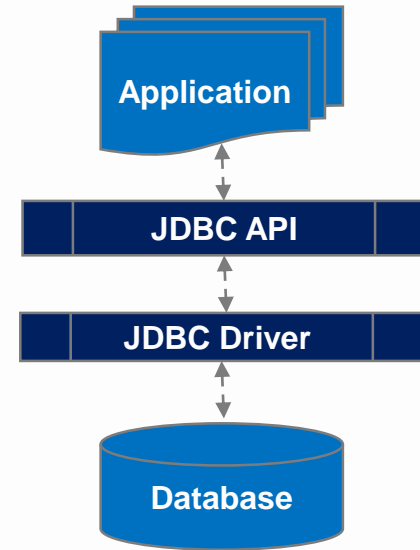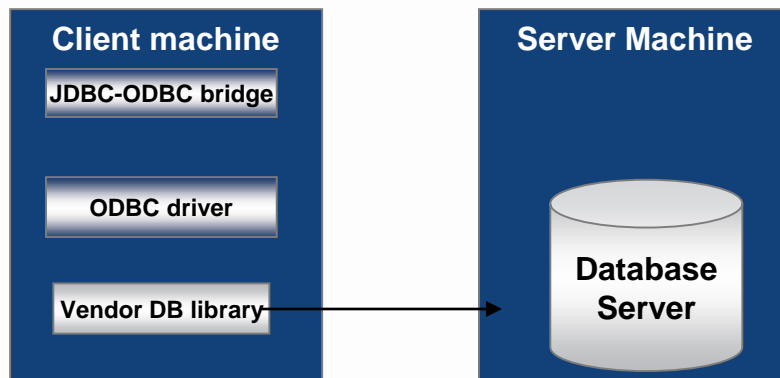
Database Drivers Types

- Type 1 JDBC-ODBC Bridge + ODBC Driver

- Type 2 Native API / Partly Java technology-enabled driver

- Type 3 Pure Java Driver for Database Middleware

- Type 4 Direct to Database Pure Java Driver

Type 1 Driver
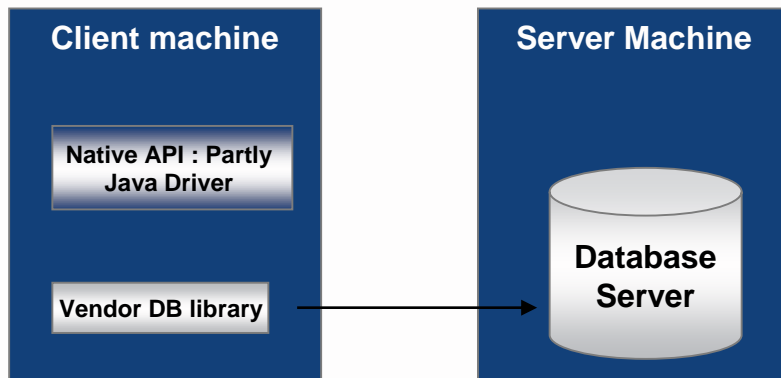
- JDBC – ODBC Bridge
  - Translates all JDBC calls into ODBC (Open Database Connectivity)

- Need to have ODBC client installed on the machine

Type 2 Driver

- Native-API/Partly Java driver

  - Converts JDBC calls into db-specific calls

  - Communicates directly with the db server

  - Requires some binary code be present on the client machine.

  - Better performance than type 1 driver

Type 3 Driver

- Net-protocol – 100% Java driver follows a three-tiered approach

  - JDBC database requests passed to the middle-tier server

  - Middle-tier server translates the request to the database-specific native-connectivity interface

    - May use a type 1 or type 2 JDBC driver

  - Request forwarded to the database server

Type 4 Driver

- Native protocol - 100% Java

- Converts JDBC calls into the vendor-specific DBMS protocol

- Client applications communicate directly with the database server

- Best performance

- Need a different driver for each database
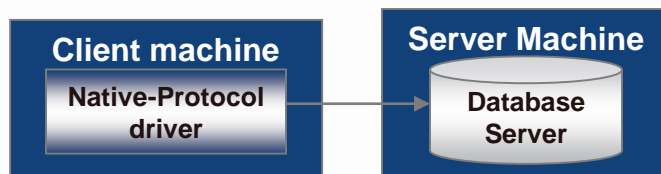
JDBC API
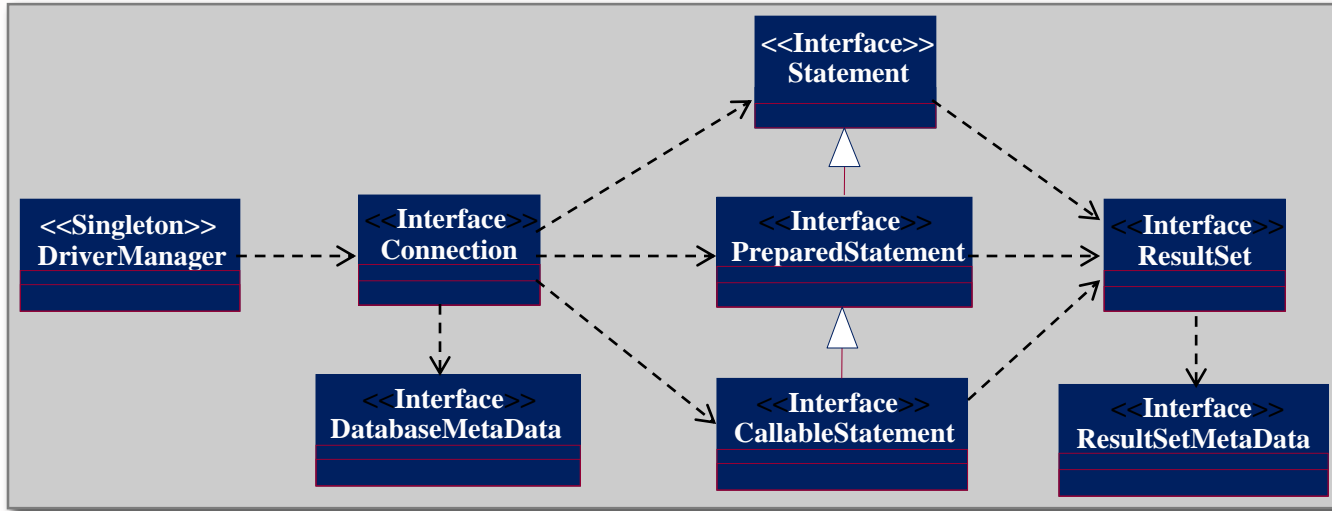
Retrieving a Connection Object

- A **Connection** object defines a connection or session with a specific database.

- It is where SQL statements are executed and results are returned.

- Before a **Connection** object can be used, the **java.sql.Connection** class has to be imported first.

LTI
Let's Solve

Retrieving a Connection Object (cont.)

- A database driver can be loaded by using the **`Class.forName()`** method

  Syntax :   *Class.forName("JDBCDriver_Name");*

  Example :   *Class.forName("oracle.jdbc.driver.OracleDriver");*

- A **`Connection`** object has three (3) important parts:
  - The URL or location of the data source
  - The username
  - The password

LTI
Let's Solve

Retrieving a Connection Object (cont.)

- A connection can be established by using the **getConnection()** method

- Creating a connection by using the URL, username and password as parameters

Connection myConnection = DriverManager.getConnection (URL, username, password);

- Creating a connection by using the URL; In this case, the URL already includes the username and password

Connection myConnection = DriverManager.getConnection (URL);

LTi
Let's Solve

Creating query statements

- A **`Statement`** is the message that the JDBC sends to the data source to either manipulate or request data from the data source

- Before a **`Statement`** object can be used, the **`java.sql.Statement`** class has to be imported first

Creating Query Statements (cont.)

- A **Statement** is created by using the following syntax:

  Syntax :   *Statement <identifier> = <Connection_Object>.createStatement();*

  Sample :   *Connection myConn = DriverManager.getConnection(...);*

  Statement myStatement = myConn.createStatement();

- A **Statement** needs to use a **Connection** object to identify which connection the statement will be associated with

LTI
Let's Solve

Creating Query Statements (cont.)

- A **Statement** can be executed by using either the **execute()**, **executeQuery()** and **executeUpdate()** methods

Syntax :   *<Statement_Object>.executeQuery("SQL Statement goes here");*

Sample :   *Connection myConn = DriverManager.getConnection(...);*

                          Statement myStatement = myConn.createStatement();

                          *myStatement.executeQuery("Select * from aTable);*

Creating Query Statements

- After a **Statement** has been executed, it returns different kinds of data depending on the type of execute used

- **execute()** returns a **boolean** value and is used to execute SQL statements written as a **String** object

- **executeUpdate()** returns an **int** value pertaining to the number of rows affected and is used to execute DDL SQL statements

- **executeQuery()** returns a **ResultSet** object and is used to execute SELECT SQL statements

Transactions

- Transaction management is vital for operations that modify or update the records in the database

- Transaction management allows the application to confirm the execution of a group of SQL statements by 'committing' the changes

- Should an exception occur, the transaction can be 'rolled-back' to undo the changes that were made

- Setting the *Connection.setAutoCommit()* method of the Connection object to *false* will prevent the statements from being committed until an explicit *Connection.commit*() is executed by the application

- The *Connection.rollback()* method can be executed should an event occur requiring the changes made so far to be undone

LTI

Let's Solve

Using PreparedStatement

- **`PreparedStatement`** is a Java object, which represents a precompiled SQL statement

- **`PreparedStatement`** is usually used for SQL statements that take parameters, although it can also execute SQL statements that have no parameters

- Before a Statement object can be used, the **`java.sql.PreparedStatement`** class has to be imported first

LTI
Let's Solve

Using PreparedStatement (cont.)

- A **PreparedStatement** is useful when an SQL statement has to be executed multiple times

- A **PreparedStatement** is precompiled, hence it is immediately executed by the DBMS, unlike **Statement,** which has to be compiled first

Syntax :

PreparedStatement <identifier> = <connection>.prepareStatement("<SQL Statement goes here>");

Example :

Connection myConn = DriverManager.getConnection(…);

*PreparedStatement pStmt = myConn.prepareStatement("Select * from Dogs Where breed = ?");*

LTI

Let's Solve

Using PreparedStatement (cont.)

- The question mark (?) serves as a wildcard or parameter and can be replaced with a value

- The wildcard can be replaced by using the appropriate methods for a particular data type, including but not limited to:
  - setString()
  - setInt()
  - setDouble()
  - setDate()
  - …

Java 6 JDBC 4.0 features

- Auto-loading of JDBC driver class

- Connection management enhancements

- Support for RowId SQL type

- DataSet implementation of SQL using Annotations

- SQL exception handling enhancements

- SQL XML support

- There are also other features such as improved support for large objects (BLOB/CLOB) and National Character Set Support.

Auto-Loading of JDBC Driver

- In JDBC 4.0, we no longer need to explicitly load JDBC drivers using Class.forName()

- When the method getConnection is called, the DriverManager will attempt to locate a suitable driver from among the JDBC drivers that were loaded at initialization and those loaded explicitly using the same class loader as the current application

- The DriverManager methods getConnection and getDrivers have been enhanced to support the Java SE Service Provider mechanism (SPM)

- According to SPM, a service is defined as a well-known set of interfaces and abstract classes, and a service provider is a specific implementation of a service. It also specifies that the service provider configuration files are stored in the META-INF/services directory

LTi
Let's Solve

Auto-Loading of JDBC Driver (Cont'd...)

- JDBC 4.0 drivers must include the file

    META-INF/services/java.sql.Driver

This file contains the name of the JDBC driver's implementation   of java.sql.Driver

For example, to load the JDBC driver to connect to a Apache Derby database, the

META-INF/services/java.sql.Driver file would contain the following entry:

org.apache.derby.jdbc.EmbeddedDriver

Connection Management

- Prior to JDBC 4.0, we relied on the JDBC URL to define a data source connection

- Now with JDBC 4.0, we can get a connection to any data source by simply supplying a set of parameters (such as host name and port number) to a standard connection factory mechanism

- New methods were added to Connection and Statement interfaces to permit improved connection state tracking and greater flexibility when managing Statement objects in pool environments

RowId Support

- The RowID interface was added to JDBC 4.0 to support the ROWID data type which is supported by databases such as Oracle and DB2

- RowId is useful in cases where there are multiple records that don't have a unique identifier column and you need to store the query output in a Collection (such Hashtable) that doesn't allow duplicates

- We can use ResultSet's getRowId() method to get a RowId and PreparedStatement's setRowId() method to use the RowId in a query

- It shouldn't be shared between different Connection and ResultSet objects

LTI

Let's Solve

Java Fundamentals

Core Java API : JDBC : Quiz

A Larsen & Toubro
Group Company

Which of the following exception must be handled while loading JDBC Driver using Class.forName(. . .) method?

SQLException

ClassNotFoundException

ClassCastException

ClassLoaderException

LTI
Let's Solve

Java Fundamentals

Unit Testing

Introduction to JUnit

- Testing is the process of checking the functionality of the application whether it is working as per requirement shared

- Junit is the DeFacto framework for developing unit test in Java

- Promotes the idea of "first testing then coding"

- Unit testing is the testing of single entity (class or method)

- A Unit Test Case is a part of code which ensures that the another part of code (method) works as expected

- There must be at least two test cases for each requirement: one positive test and one negative test

JUnit

- JUnit tests
  - "substitute the use of main() to check the program behavior"
  - All we need to do is write methods annotated with @Test annotation for writing various test cases

```
import org.junit.*;
import static org.junit.Assert.*;

public class name {
    ...

    @Test
    public void name() {   // a test case method
        ...
    }
}
```

Brief History of Java

assertTrue(**test**)	fails if the boolean test is `false`
assertFalse(**test**)	fails if the boolean test is `true`
assertEquals(**expected, actual**)	fails if the values are not equal
assertSame(**expected, actual**)	fails if the values are not the same (by ==)
assertNotSame(**expected, actual**)	fails if the values *are* the same (by ==)
assertNull(**value**)	fails if the given value is *not* `null`
assertNotNull(**value**)	fails if the given value is `null`
fail()	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:

  - e.g. `assertEquals(`**"message",** `expected, actual)`

- Why is there no `pass` method?

LTI
Let's Solve

ArrayIntList JUnit test
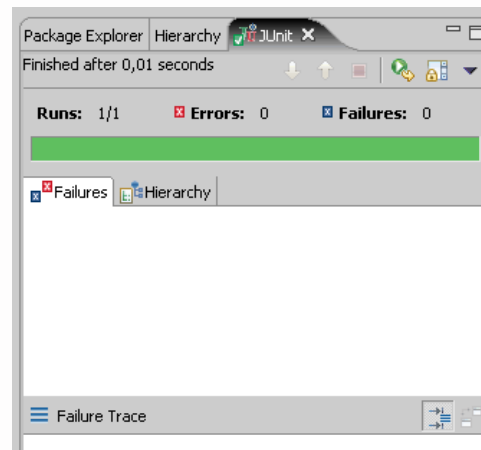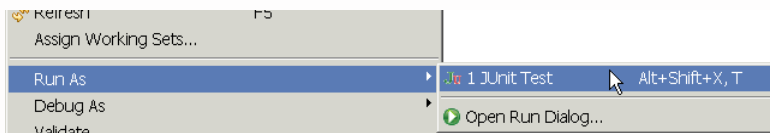
```java
import org.junit.*;
import static org.junit.Assert.*;

public class TestArrayIntList {
    @Test
    public void testAddGet1() {
        ArrayIntList list = new ArrayIntList();
        list.add(42);
        list.add(-3);
        list.add(15);
        assertEquals(42, list.get(0));
        assertEquals(-3, list.get(1));
        assertEquals(15, list.get(2));
    }

    @Test
    public void testIsEmpty() {
        ArrayIntList list = new ArrayIntList();
        assertTrue(list.isEmpty());
        list.add(123);
        assertFalse(list.isEmpty());
        list.remove(0);
        assertTrue(list.isEmpty());
    }
}
```

Running a test

- Right click it in the Eclipse Package Explorer at left; choose:

    Run As → JUnit Test

- The JUnit bar will show **green** if all tests pass, **red** if any fail.

- The Failure Trace shows which tests failed, if any, and why.

Java Fundamentals

Unit Testing :Quiz

Which of the following method of Assert class checks that a condition is true?

void assert(boolean condition)

void assertTrue(boolean condition)

void assertCheck(boolean condition)

void assertChecks(boolean condition)

Java Fundamentals

Reflection API

Introduction

- Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. Reflection API can be used for different purposes:

  - Examining classes and objects

    - One can use Reflection API to collect metadata about an Object/Class at the runtime. For ex: in eclipse, when we use the contentassist feature, it shows all possible options dynamically.

  - Manipulating classes and objects

    - One can use Reflection API to dynamically instantiate objects, access fields, invoke methods, manipulate arrays, etc.. For ex: a WebServer like Tomcat uses Reflection API to dynamically execute the service() method of the corresponding servlet based on the

    - URL submitted at the runtime

java.lang.Class class

- Class instance is the entry point in the Reflection API

- For every type of object, the Java virtual machine instantiates an immutable instance of java.lang.Class which provides methods to examine runtime properties of the object including its members and type information

- To get a hold of the Class of an object, we can use any of these different approach:
  - *Class c = obj.getClass();*
  - *Class c = String.class;*
  - *Class c = Class.forName("java.lang.String");*
  - *Class c = int.class; //for using primitives in Reflection API*

LTI

Let's Solve

Collecting information

- After we get hold of a Class instance, we can now start introspecting the target class by using the different methods provided by Class class:

  - *Method[] methods = c.getMethods();*

  - *Constructor[] constructors = c.getConstructors();*

  - *Field[] fields = c.getFields();*

- Similarly many other methods are available to collect information about the package, parent class, implemented interfaces and annotations used

Brief History of Java

```java
class Calculator {
public int add(int x, int y) {
return x + y;
}
}

Class calc = Calculator.class;
Object calcObj = calc.newInstance();
Method addMethod =
calc.getMethod("add", int.class, int.class);
Object retValue = addMethod.invoke(calcObj, 10, 20);
System.out.println(retValue);
```

▪ This is one of the most important feature of Reflection API, the power to invoke operations dynamically. We can invoke constructors, access fields, invoke methods, dynamically

LTI
Let's Solve

Which of the following is not a class of reflection API?

Field

Method

Class

Constructor

LTI
Let's Solve

Java Fundamentals

Important language features

Inner and Anonymous Classes

- An inner class is a class declared within another class and can be assigned an access modifier just like other class fields

- An inner class has private access to the other fields of a class

- There are two additional types of inner classes:
  - *local inner class* – an inner class within the body of a method
  - *anonymous inner classes* – an inner class within the body of a method without naming it

LTI
Let's Solve

Generics Overview

- *Generics* allow classes or methods to declare during compile time what type of objects they are meant to work with without having the class/method definition

- Generics is one way of avoiding lengthy type casts in order to convert objects form one class to another

LTI
Let's Solve

Generics Overview (Cont'd…)

```java
public class GenericSample <E>{
    private E field;
    public void setValue(E arg){
        this.field =arg;
    }
    public E getValue(){
        return field;
    }
}
```

- The following instantiates the class parameterized with a String

GenericSample<String> g1 = new GenericSample<String>();

Generics Overview (Cont'd…)

- That particular instance will be created as if *GenericSample* had *String* in all places where *E* was placed

- Many collections have been updated to take advantage of generics instead of using the generic *Object* class

List<String> stringList = new ArrayList<String>;

stringList.add("Hello");

stringList.add(new Integer(1)); *//Illegal now with generics, but previously legal without using generics*

LTI
Let's Solve

Bounded Type Parameters

- Wildcard notations can be used to indicate the boundaries in a class hierarchy

- The notation *<E extends Class Type>* indicates that type is a subclass of *Class Type*

```java
//Allows only subclasses of "Number" to be passed
public class GenericSample <E extends Number>{
    private E field;
    public void setValue(E arg){
        this.field =arg;
    }
    public E getValue(){
        return field;
    }
}
```

LTI
Let's Solve

WildCards

- The notation ? can be used to indicate an unknown class. This is similar to using *Object* as a parameter

- The notation <? *extends ClassType>* indicates a type that is a subclass of *ClassType*

- The notation <? super *ClassType>* indicates a type that is are parent classes of *ClassType*

Enhanced For-Loop

- *Enhanced For-Loop* reduces the need for iterators when going through an array or a collection

- It allows you to iterate through a collection without having to create an Iterator or without having to calculate beginning and end conditions for a counter variable

```
int[] sqr = {0,1,4,9,16,25};
for (int i : sqr) {
   System.out.printf(ii);
}
}
```

Autoboxing

- *Autoboxing* eliminates the drudgery of manual conversion between primitive types (such as int) and wrapper types (such as Integer).

- Any primitive variable can accept an instance whose type is the corresponding primitive wrapper.

- Any primitive wrapper reference variable can accept the corresponding primitive value as a value;

```
Float f = 9.9f;
int i = new Integer(99); //unboxing
Integer y = 20; //boxing
```

Variable Arguments (Varargs)

- Variable arguments allow methods to take in multiple arguments of the same type without specifying the number of arguments during runtime

```java
public void listNames(String… names) {
    for (String i : names) {
    System.out.println(i);
    }
}
```

- The method will be able to accept a series of Strings as a parameters that will be converted into an array

LTI
Let's Solve

Typesafe enums

- An *enum type* is a type whose *fields* consist of a fixed set of constants

- The most simple use is to list a series of constants and contained by an *enum* type

```java
enum Seasons{
    SPRING, SUMMER, FALL, WINTER
}

for(Seasons s : Seasons.values()){
System.out.println(s);
}
```

Let's Solve

Typesafe Enums

- An *enum* class is a full-fledged class that can be given fields and behaviors just like any other class.

- The constants declared can be initialized through a private or default constructor.

```java
public enum EnumSample {
    ONE(1),TWO(2),THREE(3),FOUR(4);
    private int x;

    private EnumSample(int x){
        this.x = x;
    }

    public int getX(){
        return x;
    }
}
```

Let's Solve

Static Imports

- A *static import* declaration enables programmers to refer to imported static members as if they were declared in the class that uses them

- A *static import* declaration has two forms:
  - Single static import - imports a particular static member
  - Static import on demand – imports all static members of a class

Annotations

- *Annotations* provide data about a program that is not part of the program itself

- Annotations can be used to pass information to the compiler about the source code

- Various software tools can use annotations to generate additional information, resources, or code based on the annotation

LTI
Let's Solve

Annotations Syntax

- Annotations are marked by the '@' symbol and by convention belong in their own line

  @Override

- Annotations can be given a single 'value'

  @SuppressWarnings("unchecked")

  @SuppressWarnings(value = "unchecked")

- Annotations can be given multiple named-value pairs

  @Author(

  name = "John Doe",

  date = "7/23/2008")

LTI

Let's Solve

Compiler Annotations

- *@Override*
  - Indicates that the annotated method is required to override a method from its parent class
  - The compiler will generate errors when the superclass method being annotated is not overridden

- *@Deprecated*
  - Marks a certain target element, like a method, as deprecated or outdated
  - The compiler will generate warnings when source code that calls deprecated methods is compiled

- *@SuppressWarnings*
  - Used to tell the compiler to ignore warnings like 'deprecation'

Defining Annotations

- Custom annotations are similar to declaring a class/interface source

- Custom annotations are defined by using the '@' symbol followed by the *interface* keyword

- These can be accessed by the source code using imports just like any other class or interface

Defining Annotations (cont.)

- Many annotations have no elements except the annotation name itself

  public @interface MyAnnotation{}

- Single element annotation possess a single element

  public @interface MyAnnotation{

  String myStringValue();

  }

- Multi-value annotations can have multiple elements

  public @interface MyAnnotation{

  String myStringValue();

  int myIntValue();

  }

Defining Annotations (Cont'd...)

- Method declarations should not have any parameters

- Method declarations should not have any throws clauses

- Return types are limited to the following:
  - Primitives
  - String
  - Class
  - Enum
  - Array of the above types

LTI

Let's Solve

Defining Annotations (Cont'd...)

- @Target – Defines what elements an annotation can be applied to:
  - @Target(ElementType.TYPE)—can be applied to a class or interface
  - @Target(ElementType.FIELD)—can be applied to a field or property
  - @Target(ElementType.METHOD)—can be applied to a method level annotation
  - @Target(ElementType.PARAMETER)—can be applied to the parameters of a method
  - @Target(ElementType.CONSTRUCTOR)—can be applied to constructors
  - @Target(ElementType.LOCAL_VARIABLE)—can be applied to local variables
  - @Target(ElementType.ANNOTATION_TYPE)—indicates that the declared type itself is an annotation type

- @Retention – Defines how long an annotation is maintained
  - RetentionPolicy.SOURCE: source level and will be ignored by the compiler
  - RetentionPolicy.CLASS: retained by the compiler at compile time, but ignored by the VM
  - RetentionPolicy.RUNTIME: retained by the VM so they can be read only at run-time

LTI
Let's Solve

Which of the following symbol is used to represent an annotation ?

#

%

@

&