

# Advanced Java



**Rajeev Gupta**

Java Trainer & Consultant (MTech CS)

[rgupta.mtech@gmail.com](mailto:rgupta.mtech@gmail.com)



## Rajeev Gupta

FreeLancer Corporate Java JEE/ Spring Trainer

freelance • Institution of Electronics and Telecommunication Engineers IETE

New Delhi Area, India • 500+ 

- 
1. Expert trainer for Java 8, GOF Design patterns, OOAD, JEE 7, Spring 5, Hibernate 5, Spring boot, microservice, netflix oss, Spring cloud, angularjs, Spring MVC, Spring Data, Spring Security, EJB 3, JPA 2, JMS 2, Struts 1/2, Web service
  2. Helping technology organizations by training their fresh and senior engineers in key technologies and processes.
  3. Taught graduate and post graduate academic courses to students of professional degrees.

I am open for advance java training /consultancy/ content development/ guest lectures/online training for corporate / institutions on freelance basis



**Patience, persistence and  
perspiration make an unbeatable  
combination for success.**

Napoleon Hill

## Advance Core Java Training

---

Duration: 4 Full day

prerequisite: Core Java Fundamentals

### Day 1:

- **Java Class Design fundamentals**
  - Overview OOPs concepts
  - Implement encapsulation
  - Implement inheritance including visibility modifiers and composition
  - Implement polymorphism
  - Override hashCode, equals, and toString methods from Object class
  - Create and use singleton classes and immutable classes
  - Develop code that uses static keyword on initialize blocks, variables, methods, and classes
- **Advanced Java Class Design**
  - Develop code that uses abstract classes and methods
  - Develop code that uses the final keyword
  - Create inner classes including static inner class, local class, nested class, and anonymous inner class
  - Use enumerated types including methods, and constructors in an enum type
  - Develop code that declares, implements and/or extends interfaces and use the @Override annotation
- **Generics and Collections**
  - Create and use a generic class
  - Template classes and methods
  - <? extends XXXX> <? super XXXX> <?>
  - Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects
  - Use java.util.Comparator and java.lang.Comparable interfaces

## Day 2:

- **Introduction to Stream programming, Why i should care for it?**
  - functional interface, examples of functional interface
  - @FunctionalInterface
  - Exploring existing functional interface in JDK
  - Methods vs. Functions, understanding immutability
  - Interface evolution, static method and default method inside interface
  - Issues with diamond problem interface java 8, rules to resolve it
- **What is lambda expression?**
  - Lambdas VS Anonymous Inner Classes
  - performance, how internally lambda expression works?
- **Passing code with behavior parameterization**
  - introduction to stream processing
  - Collections Streams and Filters
  - Iterate using forEach methods of Streams and List
  - Describe Stream interface and Stream pipeline
  - Filter a collection by using lambda expressions
  - Use method references with Streams
- **Lambda Built-in Functional Interfaces**
  - Use the built-in interfaces included in the java.util.function package such as
  - Predicate, Consumer, Function, and Supplier
  - Develop code that uses primitive versions of functional interfaces
  - Develop code that uses binary versions of functional interfaces
  - Develop code that uses the UnaryOperator interface

### **Day 3:**

- **Java Stream API**

- Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method
- Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch
- Develop code that uses the Optional class
- Develop code that uses Stream data methods and calculation methods
- Sort a collection using Stream API
- Save results to a collection using the collect method and group/partition data using the Collectors class
- Use flatMap() methods in the Stream API

- **Exceptions and Assertions**

- Use try-catch and throw statements
- Use catch, multi-catch, and finally clauses
- Use Autoclose resources with a try-with-resources statement
- Create custom exceptions and Auto-closeable resources
- Test invariants by using assertions

- **Use Java SE 8 Date/Time API**

- Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration
- Work with dates and times across timezones and manage changes resulting from daylight savings including Format date and times values
- Define and create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit

## Day 4:

- **Java I/O Fundamentals**
  - Read and write data from the console
  - Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package.
- **Java File I/O (NIO.2)**
  - Use Path interface to operate on file and directory paths
  - Use Files class to check, read, delete, copy, move, manage metadata of a file or directory
  - Use Stream API with NIO.2
- **Java Concurrency**
  - Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks
  - Identify potential threading problems among deadlock, starvation, livelock, and race conditions
  - Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution
  - Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayListUse parallel Fork/Join Framework
  - Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.

- **Java Concurrency**
  - Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks
  - Identify potential threading problems among deadlock, starvation, livelock, and race conditions
  - Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution
  - Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayListUse parallel Fork/Join Framework
  - Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.
- **Building Database Applications with JDBC**
  - Describe the interfaces that make up the core of the JDBC API including the Driver, Connection, Statement, and ResultSet interfaces and their relationship to provider implementations
  - Identify the components required to connect to a database using the DriverManager class including the JDBC URL
  - Submit queries and read results from the database including creating statements, returning result sets, iterating through the results, and properly closing result sets, statements, and connections
- **Localization**
  - Read and set the locale by using the Locale object
  - Create and read a Properties file
  - Build a resource bundle for each locale and load a resource bundle in an application



## Advance Core Java Training

---

Duration: 4 Full day

prerequisite: Core Java Fundamentals

### Day 1:

- **Java Class Design fundamentals**
  - Overview OOPs concepts
  - Implement encapsulation
  - Implement inheritance including visibility modifiers and composition
  - Implement polymorphism
  - Override hashCode, equals, and toString methods from Object class
  - Create and use singleton classes and immutable classes
  - Develop code that uses static keyword on initialize blocks, variables, methods, and classes
- **Advanced Java Class Design**
  - Develop code that uses abstract classes and methods
  - Develop code that uses the final keyword
  - Create inner classes including static inner class, local class, nested class, and anonymous inner class
  - Use enumerated types including methods, and constructors in an enum type
  - Develop code that declares, implements and/or extends interfaces and use the @Override annotation
- **Generics and Collections**
  - Create and use a generic class
  - Template classes and methods
  - <? extends XXXX> <? super XXXX> <?>
  - Create and use ArrayList, TreeSet, TreeMap, and ArrayDeque objects
  - Use java.util.Comparator and java.lang.Comparable interfaces

# What is Java?

Java=OOPL+JVM+lib

- How Java is different then C++?

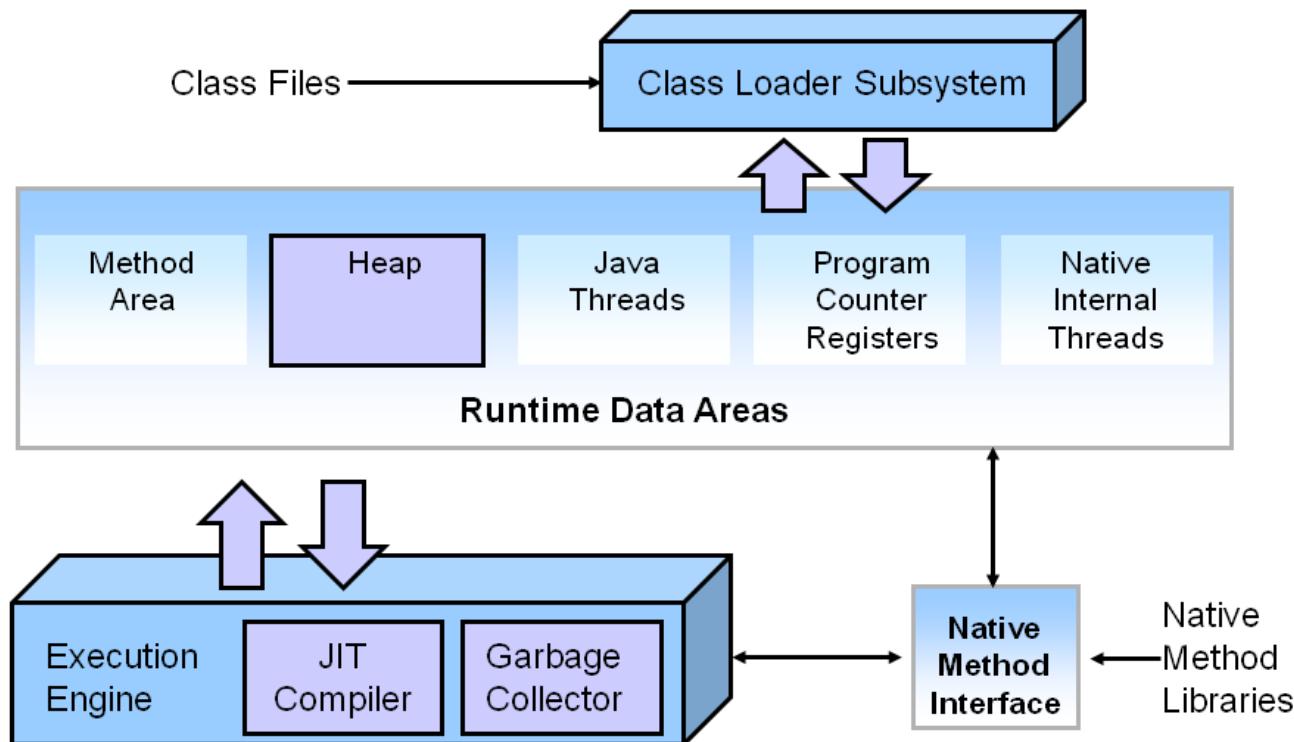


Java Language		Java Language													
Tools & Tool APIs		java		javac		javadoc		apt		jar					
Deployment Technologies		Security		Int'l		RMI		IDL		Deploy					
User Interface Toolkits		Deployment				Java Web Start				Java Plug-in					
Integration Libraries		AWT				Swing				Java 2D					
JDK		Accessibility		Drag n Drop		Input Methods		Image I/O		Print Service					
JRE		IDL		JDBC™		JNDI™		RMI		RMI-IIOP					
Other Base Libraries		Beans		Intl Support		I/O		JMX		JNI					
lang and util Base Libraries		Networking		Override Mechanism		Security		Serialization		Extension Mechanism					
lang and util Base Libraries		lang and util		Collections		Concurrency Utilities		JAR		Logging					
Java Virtual Machine		Preferences API		Ref Objects		Reflection		Regular Expressions		Versioning					
Platforms		Java Hotspot™ Client VM				Java Hotspot™ Server VM									
Platforms		Solaris™			Linux		Windows			Other					



# Understanding JVM

## Key HotSpot JVM Components



### Search class file in <JAVA\_HOME>/jre/lib (Bootstrap Class Loader)

- 1. loadClass() method will be called by passing fully qualified name of name.
- 2. It will call findLoadedClass() method to check if class is already loaded in class and delegates request to parent class if it not loaded.
- 1. If parent class loader is able to locate requested class it will load that.
- 2. If parent class loader is not able load or find class it will delegate request to child class.

### Search class file in <JAVA\_HOME>/jre/lib/ext (Extension class loader)

- 1. loadClass() method will be called by passing fully qualified name of name
- 2. It will call findLoadedClass() method to check if class is already loaded in class and delegates request to parent class if it not loaded.
- 1. If parent class loader is able to locate requested class it will load that.
- 2. If parent class loader is not able load or find class it will delegate request to child class.

Request for a .class file

Search class file in class path  
(System Class Loader)

Exception will be thrown

# Object technologies

- OO is a way of looking at a software system as a collection of **interactive objects**

Reality



Tom's House



Tom



Tom's Car

Model



# What is an Object?

- Informally, an object represents an entity, either physical, conceptual, or software.

- Physical entity



Tom's Car

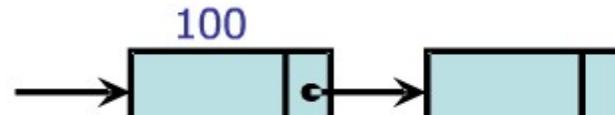
- Conceptual entity



US\$100,000,000,000

Bill Gate's bank account

- Software entity



# Class

- A class is the blueprint from which individual objects are created.
- An object is an instance of a class.



Object factory



Cookie Cutter



```
public class StudentTest {  
    public static void main(String[] args) {  
        Student s1 = new Student();  
        Student s2 = new Student();  
    }  
}
```

- class -

```
public class Student {  
    private String name;  
    // ...  
}
```

# Classes and Object

- All the objects share the same attribute names and methods with other objects of the same class
- Each object has its own value for each of the attribute

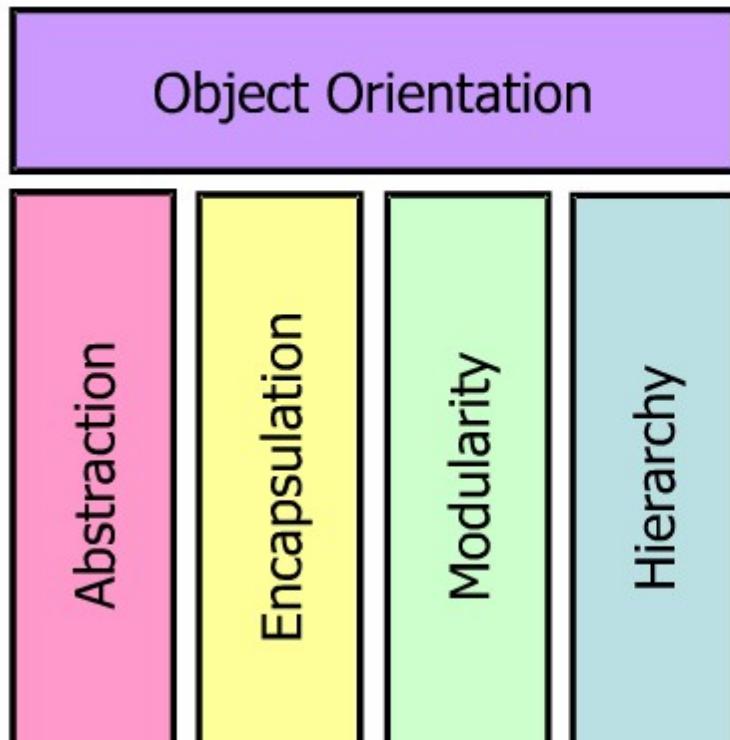
Person
- name
- dateOfBirth
+ getAge()



harry:Person	
name	Harry
dateOfBirth	8/12/1975
+ getAge()	

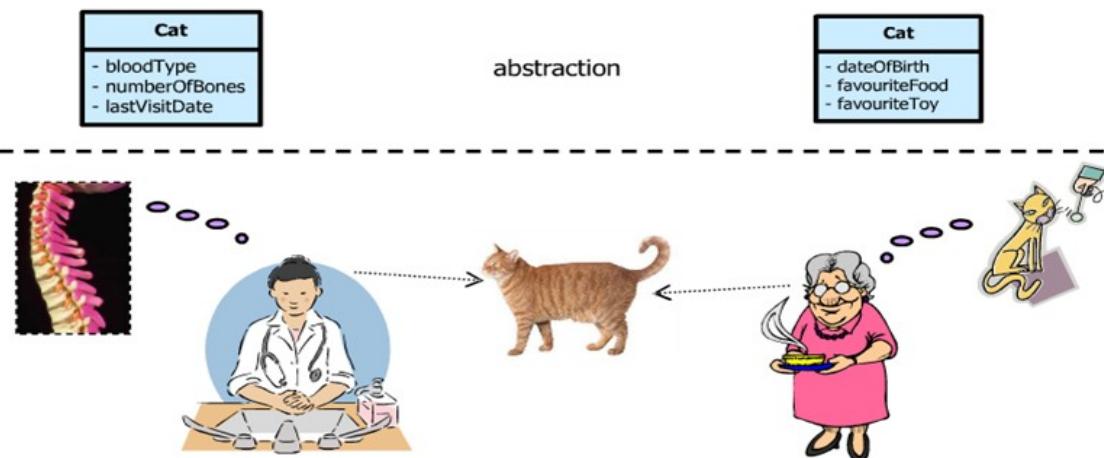
mary:Person	
name	Mary
dateOfBirth	8/12/1980
+ getAge()	

# Basic principles of OO



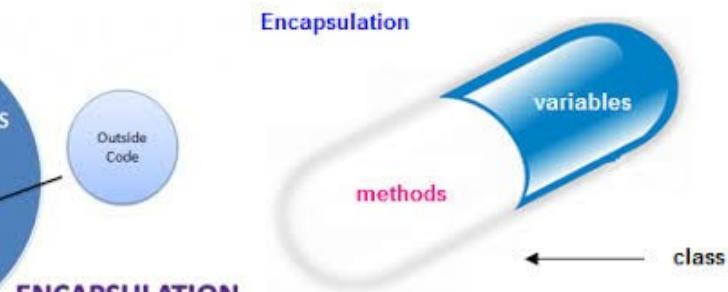
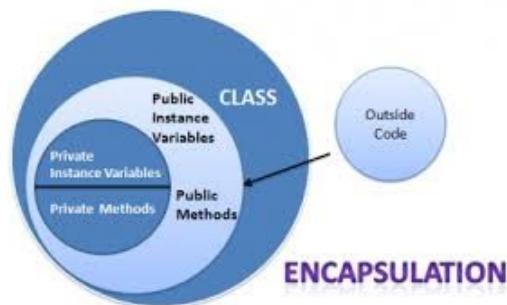
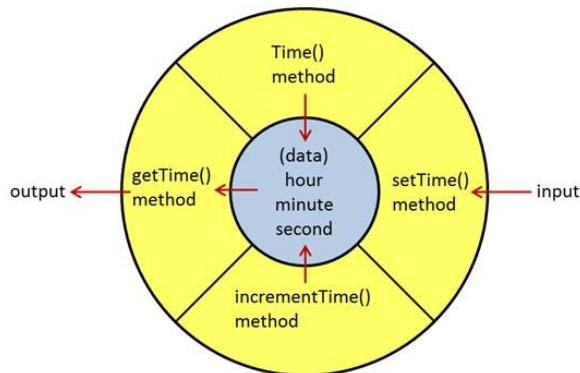
# Abstraction

- Fundamental ways that we use to cope with complexity
- "abstraction arises from a **recognition of similarities** between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to **ignore for the time being the differences**" -Hoare
- An abstraction denotes the **essential characteristics of an object that distinguish it from all other kinds of objects** and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.
  - Determine the **relevant properties** and features while ignoring non-essential details



# Encapsulation

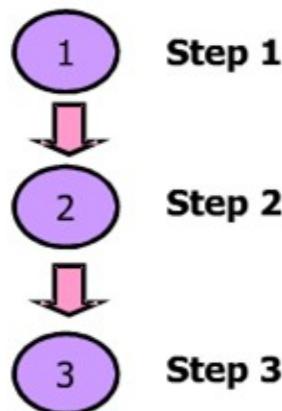
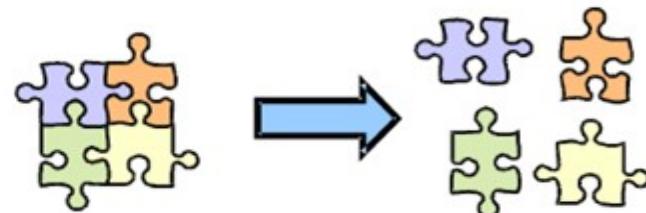
- Abstraction and encapsulation are *complementary concepts*: abstraction focuses upon the observable behavior of an object, whereas encapsulation focuses upon the implementation that gives rise to this behavior.
- Encapsulation is most often achieved through information hiding which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods.
- Information hiding is tool to achieve encapsulation



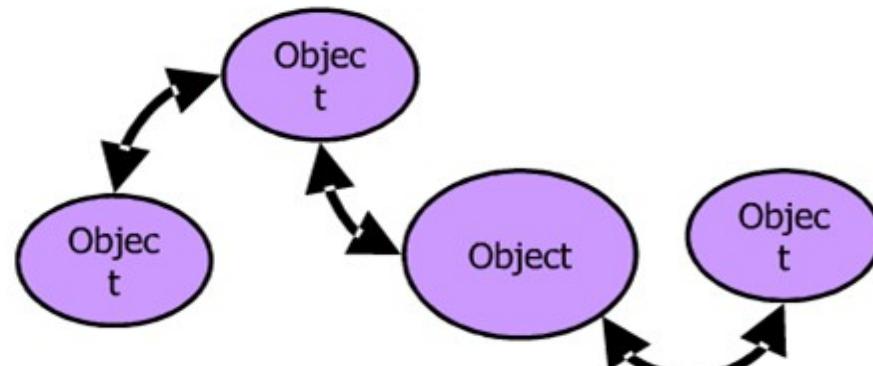
~~class Hacker{  
Account a= new Account();  
a.account\_balance= -100;  
}~~

# Modularity

- Break something complex into manageable pieces
  - Functional **Decomposition**
  - Object Decomposition



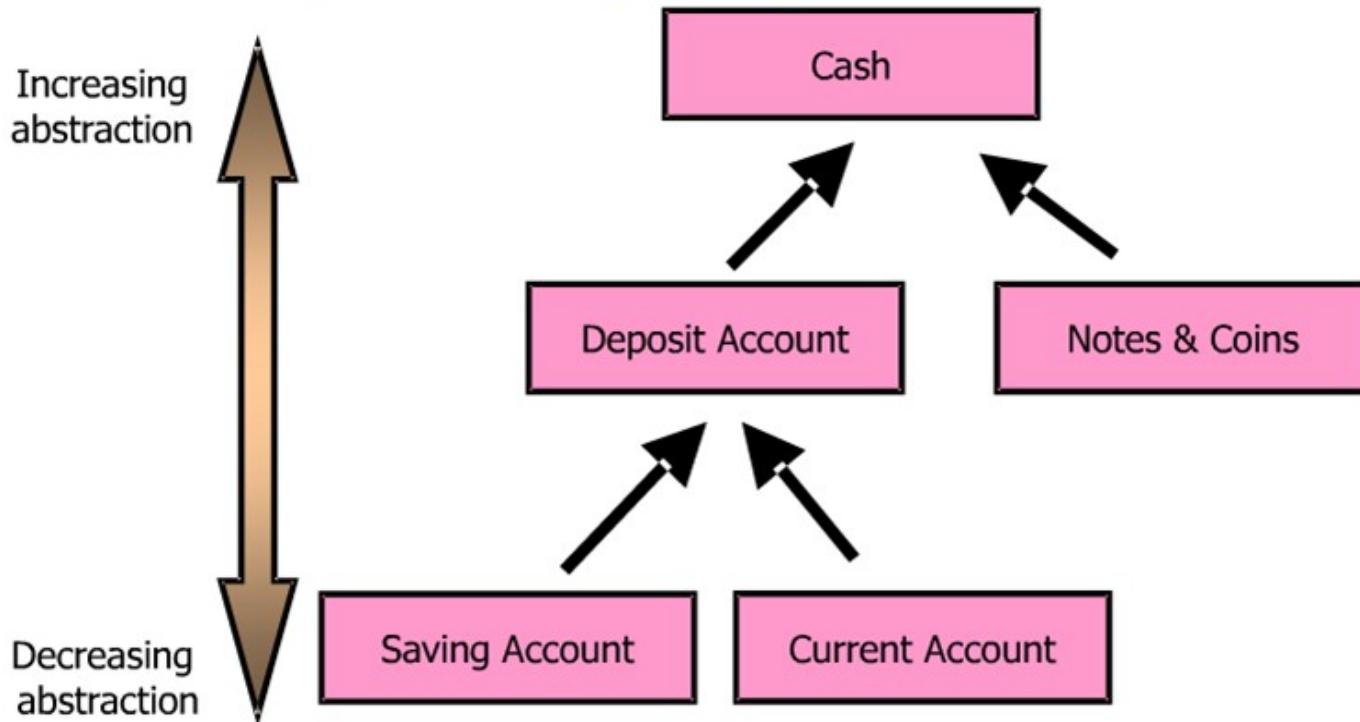
Functional Decomposition



Object Decomposition

# Hierarchy

- Ranking or ordering of objects



# Constructors: default, parameterized and copy

- Initialize state of the object
- Special method have same name as that of class
- Can't return anything
- Can only be called once for an object
- Can be private
- Can't be static\*
- Can overloaded but can't overridden\*
- Three type of constructors
  - Default,
  - Parameterized and
  - Copy constructor

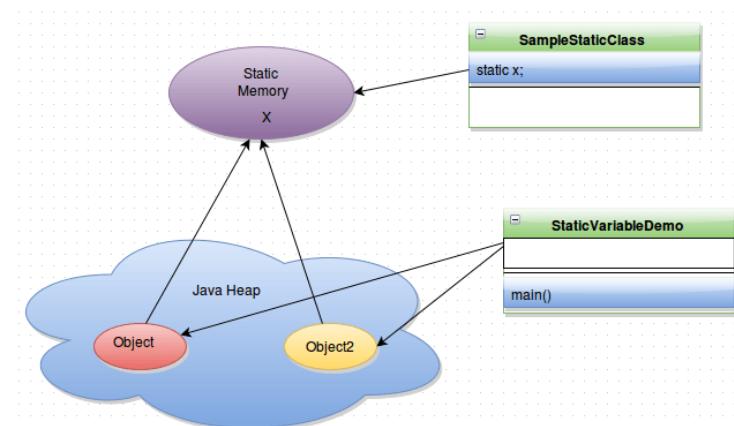
```
class Account{  
    private int id;  
    private double balance;  
  
    //default ctr  
    public Account() {  
        //.....  
    }  
  
    //parameterized ctr  
    public Account(int i, double b) {  
        this.id=i;  
        this.balance=b;  
    }  
  
    //copy ctr  
    public Account(Account ac) {  
        //.....  
    }  
}
```

# Static method/variable

- Instance variable -per object while static variable are per class
- Initialize and define before any objects
- Most suitable for counter for object
- Static method can only access static data of the class
- For calling static method we don't need an object of that class

Now guess why main was static?

How to count number of account object in the memory?



# Using static data..

```
class Account{  
  
    private int id;  
    private double balance;  
  
    // will count no of account in application  
    private static int totalAccountCounter=0; → static variable  
  
    public Account(){  
        totalAccountCounter++;  
    }  
  
    public static int getTotalAccountCounter(){  
        return totalAccountCounter; → static method  
    }  
}
```

```
Account ac1=new Account();  
Account ac2=new Account();
```

```
//How many account are there in application ?
```

```
System.out.println(Account.getTotalAccountCounter());
```

```
System.out.println(ac1.getTotalAccountCounter());
```

We can not access instance variable in static method but can access static variable in instance method

# Initialization block

- We can put repeated constructor code in an Initialization block...
- Static Initialization block runs before any constructor and runs only once...

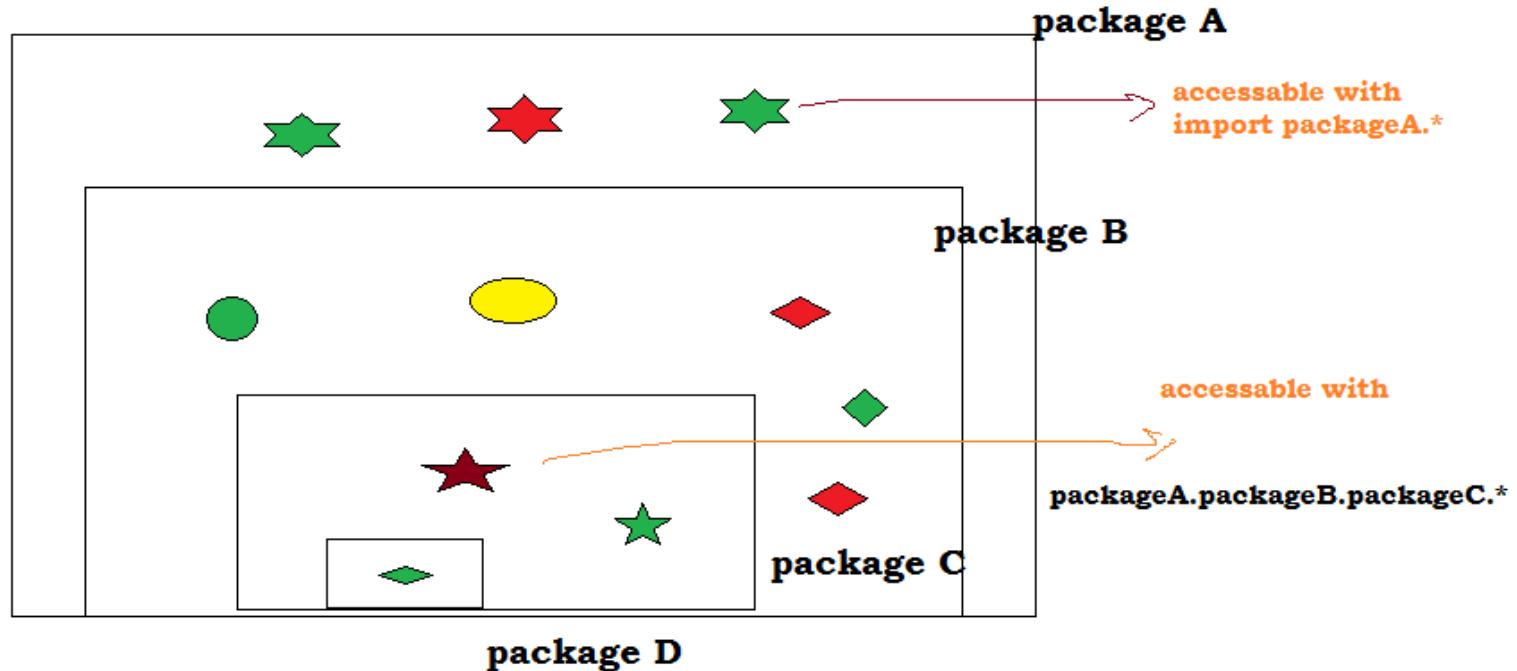
```
class Account{  
    private int id;  
    private double balance;  
    private int accountCounter=0;  
  
    static{  
        System.out.println("static block: runs only once ...");  
    }  
  
    {  
        System.out.println("Init block 1: this runs before any constructor ...");  
    }  
  
    {  
        System.out.println("Init block 2: this runs after init block 1 , before any const execute ...");  
    }  
}
```

```
class Account{  
  
    private int id;  
    private double balance;  
  
    public Account(){  
        //this is common code  
    }  
    public Account(int id , double balance){  
        //this is common code  
        this.id=id;  
        this.balance=balance;  
    }  
}
```

code repetition.

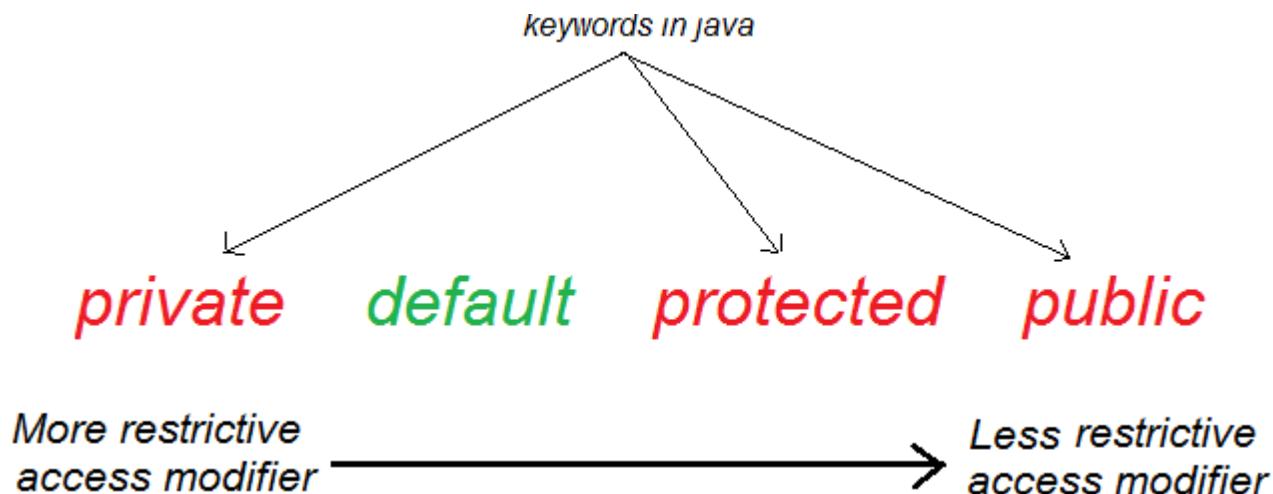
# Packages

- Packages are Java's way of grouping a number of related classes and/or interfaces together into a single unit.
- Packages act as "containers" for classes.



# Visibility Modifiers

- For instance variable and methods
  - Public
  - Protected
  - Default (package level)
  - Private
- For classes
  - Public and default



# Visibility Modifiers

- class A has default visibility hence can access in the same package only.
- Make class A public, then access it.
- Protected data can access in the same package and all the subclasses in other packages provide class itself is public

```
pack packA;
```

```
class A{  
    public void foo(){  
    }  
}
```

```
pack packB;  
import packA.*;
```

```
class B{  
    public void boo(){  
A a=new A();  
}
```

```
pack packA;
```

```
public  
class A{  
    protected  
    void foo(){  
    }  
}
```

```
pack packB;  
import packA.*;
```

```
class B{  
    public void boo(){  
A a=new A();  
}
```

```
pack packB;  
import packA.*;  
class C extends A{
```

```
    public void foo2(){  
        foo();  
    }  
}
```

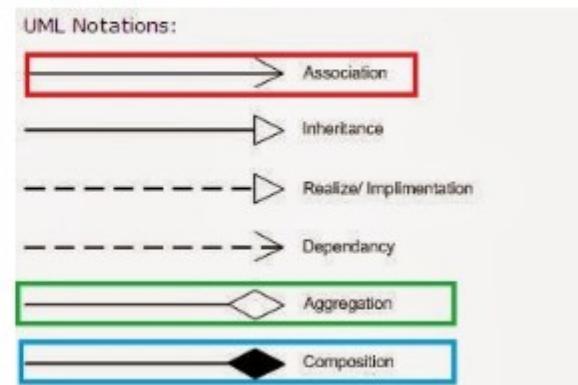
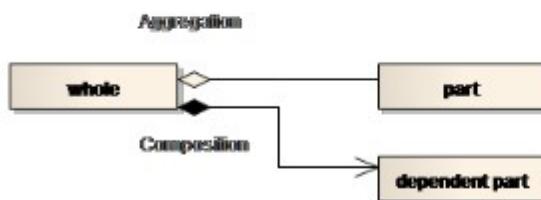
# A bit about UML diagram...

- UML 2.0 aka modeling language has 12 type of diagrams
- Most important once are class diagram, use case diagram and sequence diagram.
- You should know how to read it correctly
- This is not UML session... □

UML	Implementation
<pre>class Person {     - name : String     - age : int      + Person(name : String)     + calculateBMI() : double     + isOlderThan(another : Person) : boolean</pre>	<pre>public class Person {     private String name;     private int age;      public Person(String name) { ... }      public double calculateBMI() { ... }      public boolean isOlderThan(Person another) {...} }</pre>

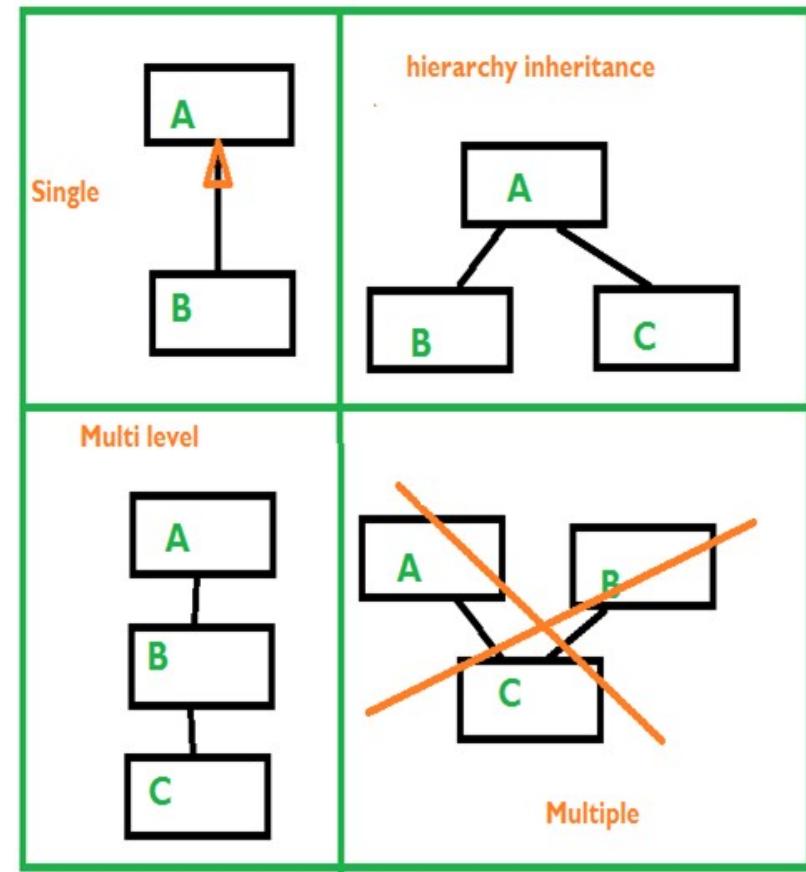
# Relationship between Objects

- USE-A
  - Passanger using metro to reach from office from home
- HAS-A (Association)
  - Compostion
    - Flat is part of Durga apartment
  - Aggregation
    - Ram is musician with RockStart musics group
- IS-A
  - Empoloyee is a person

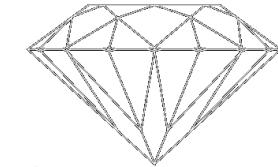


# Inheritance

- ***Inheritance is the inclusion of behaviour (i.e. methods) and state (i.e. variables) of a base class in a derived class so that they are accessible in that derived class.***
- **code reusability.**
- **Subclass and Super class concept**

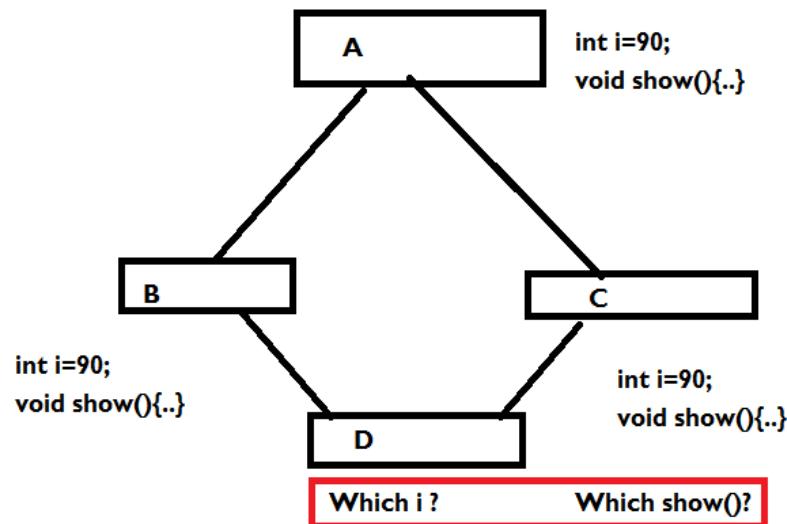


# Diamond Problem?



*Diamond*

- Hierarchy inheritance can leads to poor design..
- Java don't support it directly...( Possible using interface )



# Inheritance example

- Use extends keyword
- Use **super** to pass values to base class constructor.

```
class A{  
    int i;  
    A() {System.out.println("Default ctr of A");}  
    A(int i) {System.out.println("Parameterized ctr of A");}  
}  
  
class B extends A{  
  
    int j;  
    B() {System.out.println("Default ctr of B");}  
    B(int i,int j)  
    {  
        super(i);  
        System.out.println("Parameterized ctr of B");  
    }  
}
```

# Overloading

- Overloading deals with multiple methods in the same class with the same signatures.

```
class MyClass {  
    public void getInvestAmount(int rate) {...}  
  
    public void getInvestAmount(int rate, long principal)  
    { ... }  
}
```

- Both the above methods have the same method names but different method signatures, which mean the methods are overloaded.
- **Overloading lets you define the same operation in different ways for different data.**

## Constructor can be overloaded

**Be careful of overloading ambiguity**

rgupta.mtech@gmail.com  
**\*Overloading in case of var-arg and Wrapper objects...**

# Overriding...

- Overriding deals with two methods, one in the parent class and the other one in the child class and has the same name and signatures.
- Both the above methods have the same method names and the signatures but the method in the subclass *MyClass* overrides the method in the superclass *BaseClass*
- Overriding lets you define the **same operation in different ways for different object types.**

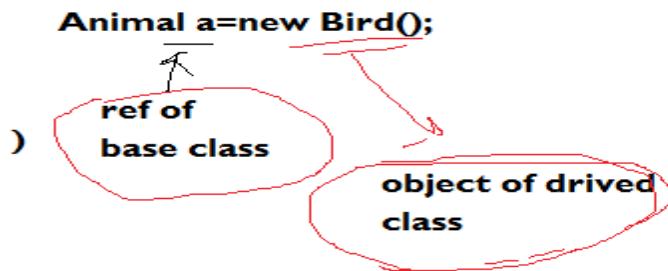
```
class BaseClass{  
    public void getInvestAmount(int rate) {...}  
}  
  
class MyClass extends BaseClass {  
    public void getInvestAmount(int rate) { ...}  
}
```

# Polymorphism

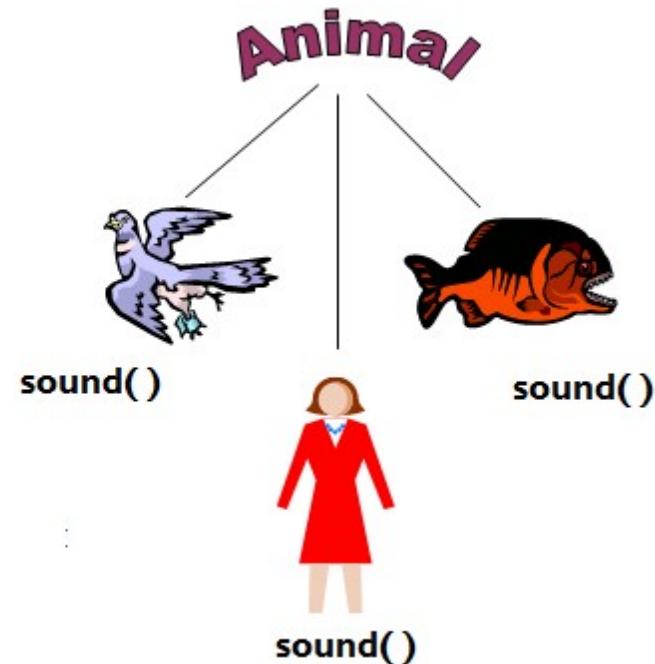
- Polymorphism=many forms of one things
- ***Substitutability***
- ***Overriding***
- ***Polymorphism means the ability of a single variable of a given type to be used to reference objects of different types, and automatically call the method that is specific to the type of object the variable references.***

# Polymorphism

- Every animal sound but differently...
- We want to have Pointer of Animal class assigned by object of derived class



Which method is going to be called is not decided by the type of pointer rather object assigned will decide at run time



# Example...

```
class Animal
{
    public void sound()
    {
        System.out.println("Don't know how generic animal sound.....");
    }
}
class Bird extends Animal
{@Override
    public void sound()
    {
        System.out.println("Bird sound.....");
    }
}
class Person extends Animal
{@Override
    public void sound()
    {
        System.out.println("Person sound.....");
    }
}
```

# Need of abstract class?

- Sound( ) method of Animal class don't make any sense ...i.e. it don't have semantically valid definition
- Method sound( ) in Animal class should be **abstract** means incomplete
- Using abstract method Derivatives of Animal class **forced** to provide meaningful sound() method

```
class Animal
{
    public void sound()
    {
        System.out.println("Don't know how generic animal sound.....");
    }
}
class Bird extends Animal
{@Override
    public void sound()
    {
        System.out.println("Bird sound.....");
    }
}
class Person extends Animal
{@Override
    public void sound()
    {
        System.out.println("Person sound.....");
    }
}
```

# Abstract class

- If an class have at least one abstract method it should be declare abstract class.
- Some time if we want to stop a programmer to create object of some class...
- Class has some default functionality that can be used as it is.
- Can extends only one abstract class □



```
class Foo{  
    public abstract void foo();  
}
```



```
abstract class Foo{  
    public abstract void foo();  
}
```



```
abstract class Foo{  
}
```

# Abstract class use cases...

- Want to have some default functionality from base class and class have some abstract functions that can't be defined at that moment

```
abstract class Animal
{
    public abstract void sound();
    public void eat()
    {
        System.out.println("animal eat...");
    }
}
```

- Don't want to allow a programmer to create object of an class as it is too generic

- Interface vs. abstract class

# More example...

```
public abstract class Account {  
    public void deposit (double amount) {  
        System.out.println("depositing " + amount);  
    }  
  
    public void withdraw (double amount) {  
        System.out.println ("withdrawing " + amount);  
    }  
  
    public abstract double calculateInterest(double amount);  
}
```

```
public class SavingsAccount extends Account {  
  
    public double calculateInterest (double amount) {  
        // calculate interest for SavingsAccount  
        return amount * 0.03;  
    }  
  
    public void deposit (double amount) {  
        super.deposit (amount); // get code reuse  
        // do something else  
    }  
  
    public void withdraw (double amount) {  
        super.withdraw (amount); // get code reuse  
        // do something else  
    }  
}
```

```
public class TermDepositAccount extends Account {  
  
    public double calculateInterest (double amount) {  
        // calculate interest for SavingsAccount  
        return amount * 0.05;  
    }  
  
    public void deposit(double amount) {  
        super.deposit (amount); // get code reuse  
        // do something else  
    }  
  
    public void withdraw(double amount) {  
        super.withdraw (amount); // get code reuse  
        // do something else  
    }  
}
```

# Final

- What is the meaning of final
  - Something that can not be change!!!
- final
  - Final method arguments
    - Cant be change inside the method
  - Final variable
    - Become constant, once assigned then cant be changed
  - Final method
    - Cant overridden
  - Final class
    - Can not inherited (Killing extendibility )
      - **Can be reuse**

Some examples....

# Final class

- Final class can't be subclass i.e. Can't be extended
  - No method of this class can be overridden
  - Ex: String class in Java...
- **Real question is in what situation somebody should declare a class final**

```
package cart;

public final class Beverage{

    public void importantMethod() {
        sysout("hi");
    }
}
```

---

~~```
package examStuff;
import cart.*;

class Tea extends beverage{
```~~

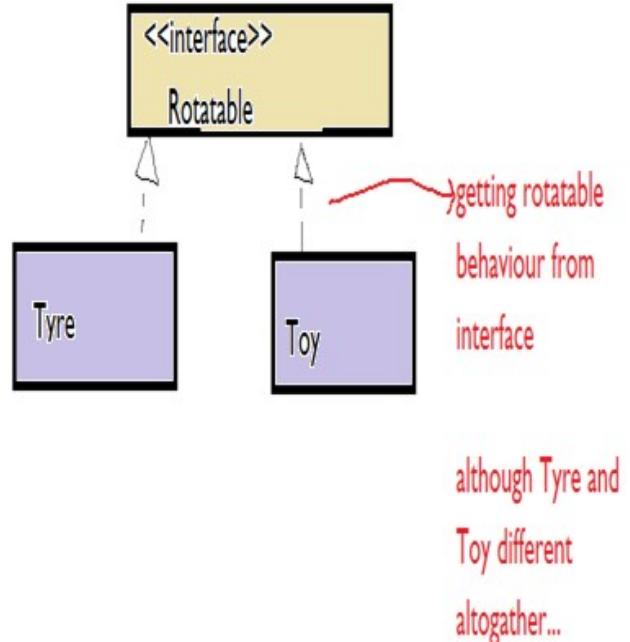
# Final Method

- Final Method Can't be overridden
- Class containing final method can be extended

```
class Foo{  
  
    public final void foo(){  
        Sysout("I am the best");  
        Sysout("You can use me but can't override me");  
    }  
};  
  
class Bar extends Foo{  
    @Override  
    public final void foo(){  
  
    }  
}
```

# Interface?

- Interface : Contract bw two parties
- Interface method
  - Only declaration
  - No method definition
- Interface variable
  - Public static and final constant
    - Its how java support global constar
- Break the hierarchy
- Solve diamond problem
- Callback in Java\*



# Interface?

- Rules
  - All interface methods are always public and abstract, whether we say it or not.
  - Variable declared inside interface are always public static and final
  - Interface method can't be static or final
  - Interface cant have constructor
  - An interface can extends other interface
  - Can be used polymorphically
  - A class implementing an interface must implement all of its method otherwise it need to declare itself as an abstract class...

# Implementing an interface...

## What we declare

```
interface Bouncable{  
    int i=9;  
    void bounce();  
    void setBounceFactor();  
}
```

## What compiler think...

```
interface Bouncable{  
    public static final int i=9;  
    public abstract void bounce();  
    public abstract void setBounceFactor();  
}
```

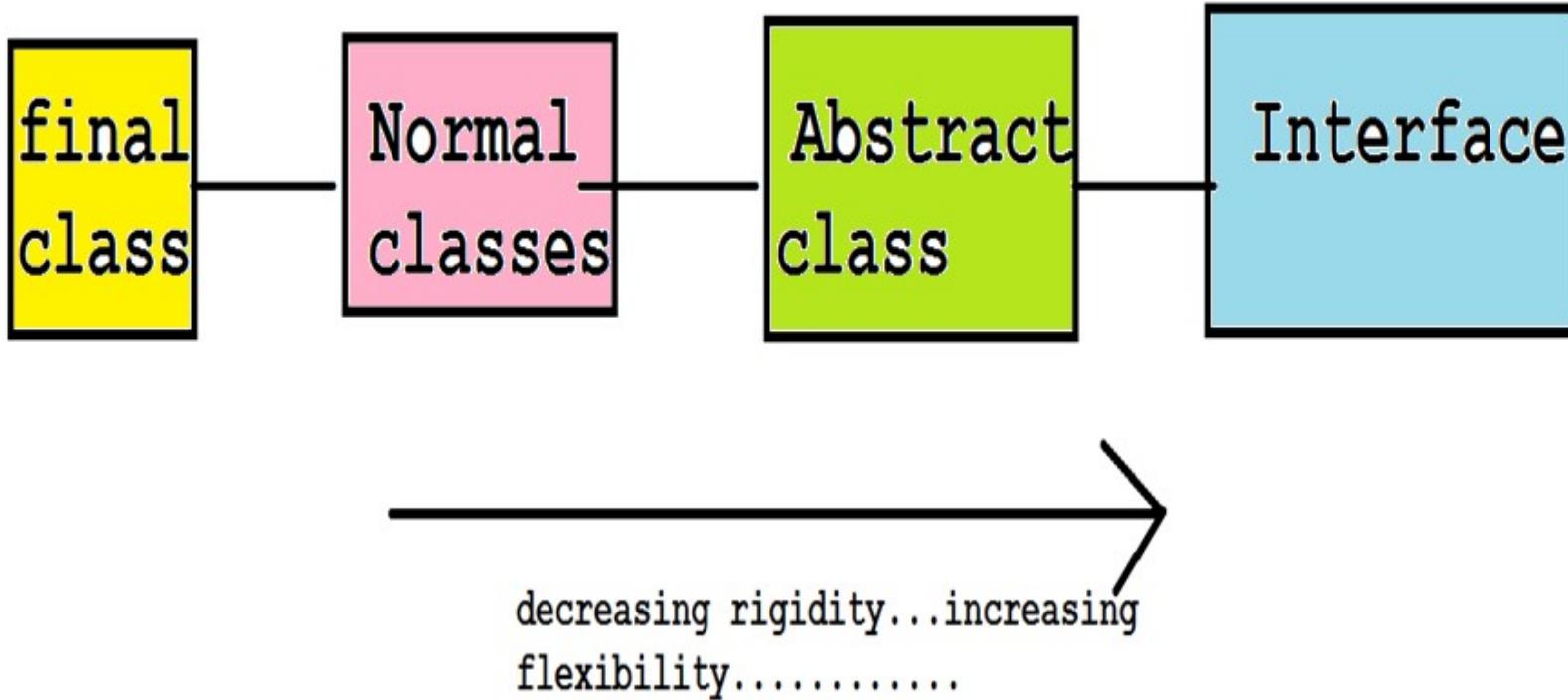
All interface method  
must be  
implemented....

```
class Tyre implements Bouncable{  
  
    public void bounce(){  
        Sysout(i);  
        Sysout(i++);  
    }  
    public void setBounceFactor(){}
}
```

# Note

- Following interface constant declaration are identical
  - int i=90;
  - public static int i=90;
  - public int i=90;
  - public static int i=90;
  - public static final int i=90;
- Following interface method declaration don't compile
  - **final** void bounce();
  - **static** void bounce();
  - **private** void bounce();
  - **protected** void bounce();

# Decreasing Rigidity..increasing Flexibility



# Type of relationship bw objects

- USE-A
- HAS-A
- IS-A (Most costly ? )

ALWAYS GO FOR LOOSE COUPLING AND HIGH COHESION...

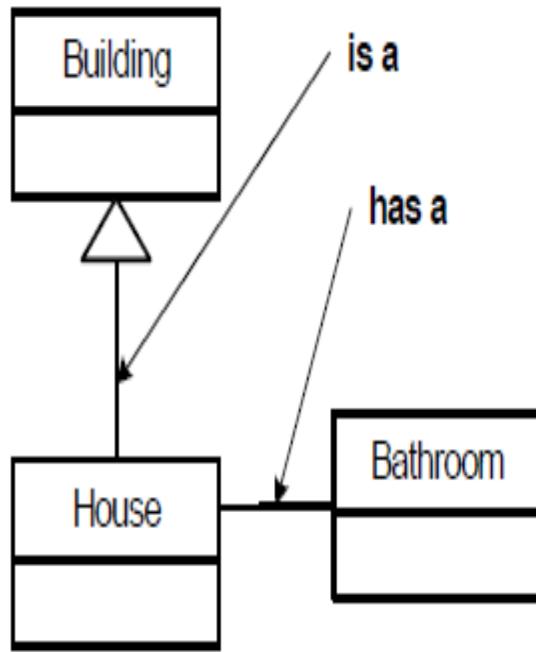
But HOW?

# IS-A

# VS

# HAS-A

## Inheritance [ is a ] Vs Composition [ has a ]



**is a** [House is a Building]

```
class Building{  
    ....  
}  
  
class House extends Building{  
    ....  
}
```

**has a** [House has a Bathroom]

```
class House {  
    Bathroom room = new Bathroom();  
    ....  
    public void getTotMirrors(){  
        room.getNoMirrors();  
        ....  
    }  
}
```

# String

- Immutable i.e. once assigned then can't be changed
- Only class in java for which object can be created with or without using **new** operator

Ex: String s="india";  
String s1=new String("india");      **What is the difference?**

- String concatenation can be in two ways:
  - String s1=s+ "paki";    **Operator overloading**
  - String s3=s1.concat("paki");

# Immutability

- Immutability means something that cannot be changed.
- Strings are immutable in Java. What does this mean?
  - String literals are very heavily used in applications and they also occupy a lot of memory.
  - Therefore for efficient memory management, all the strings are created and kept by the JVM in a place called string pool (which is part of Method Area).
  - Garbage collector does not come into string pool.
  - How does this save memory?
  - 
  - **RULES FOR IMMUTABILITY**

Declare the class as final so it can't be extended.

Make all fields private so that direct access is not allowed.

Don't provide setter methods for variables

Make all mutable fields final so that it's value can be assigned only once.

Initialize all the fields via a constructor performing deep copy.

Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

# How to make copy of an Object?

```
int[] src = new int[]{1,2,3,4,5};  
int[] dest = new int[5];  
System.arraycopy( src, 0, dest, 0, src.length );
```

```
int[] a = new int[]{1,2,3,4,5};  
int[] b = a.clone();
```

```
int[] a = {1,2,3,4,5};  
int[] b = Arrays.copyOf(a, a.length);
```

Arrays.copyOfRange():

If you want few elements of an array to be copied

```
public final class ImmutableReminder{  
    private final Date remindingDate;  
  
    public ImmutableReminder (Date remindingDate) {  
        if(remindingDate.getTime() < System.currentTimeMillis()){  
            throw new IllegalArgumentException("Can not set reminder"  
                " for past time: " + remindingDate);  
        }  
        this.remindingDate = new Date(remindingDate.getTime());  
    }  
  
    public Date getRemindingDate() {  
        return (Date) remindingDate.clone();  
    }  
}
```

## There are certain steps that are to be followed for creating an immutable class –

1. Methods of the class should not be overridden by the subclasses. You can ensure that by making your class **final**.
2. Make all **fields final and private**. If the field is of **primitive type**, then its value can't be changed as it is final. If field is holding a reference to another object, then declaring that field as final means its reference can't be changed.  
Having **access modifier** as private for the fields ensure that fields are not accessed outside the class.
3. Initialize all the fields in a **constructor**.
4. Don't provide setter methods or any method that can change the state of the object.  
Only provide methods that can access the value (like getters).
5. In case any of the fields of the class holds reference to a mutable object any change to those objects should also not be allowed, for that –
  - Make sure that there are no methods within the class that can change those mutable objects (change any of the field content).
  - Don't share reference of the mutable object, if any of the methods of your class return the reference of the mutable object then its content can be changed.
  - If reference must be returned create copies of your internal mutable objects and return those copies rather than the original object. The copy you are creating of the internal mutable object must be a **deep copy** not a shallow copy.

# String comparison

- Two string should never be checked for equality using == operator  
**WHY?**
- Always use equals( ) method....

```
String s1="india";
String s2="paki";
```

```
if(s1.equals(s2))
```

```
....
```

```
....
```

|              | String               | StringBuffer   | StringBuilder  |
|--------------|----------------------|----------------|----------------|
| Storage Area | Constant String Pool | Heap           | Heap           |
| Modifiable   | No (immutable)       | Yes( mutable ) | Yes( mutable ) |
| Thread Safe  | Yes                  | Yes            | No             |
| Performance  | Fast                 | Very slow      | Fast           |

## Various memory area of JVM

1. Method area ----- Per JVM
2. heap area ----- Per JVM
3. stack area ----- Per thread
4. PC register area ----- Per thread
5. Native method area ----- Per thread

### String

**String** is **immutable**: you can't modify a string object but can replace it by creating a new instance. Creating a new instance is rather expensive.

```
//Inefficient version using immutable String
String output = "Some text"
int count = 100;
for(int i=0; i<count; i++) {
    output += i;
}
return output;
```

The above code would build 99 new **String** objects, of which 98 would be thrown away immediately. Creating new objects is not efficient.

### StringBuffer / StringBuilder (added in J2SE 5.0)

**StringBuffer** is **mutable**: use **StringBuffer** or **StringBuilder** when you want to modify the contents. **StringBuilder** was added in Java 5 and it is identical in all respects to **StringBuffer** except that it is not synchronized, which makes it slightly faster at the cost of not being thread-safe.

```
//More efficient version using mutable StringBuffer
StringBuffer output = new StringBuffer(110)// set an initial size of 110
output.append("Some text");
for(int i=0; i<count; i++) {
    output.append(i);
}
return output.toString();
```

The above code creates only two new objects, the **StringBuffer** and the final **String** that is returned. **StringBuffer** expands as needed, which is costly however, so it would be better to initialize the **StringBuffer** with the correct size from the start as shown.

# Enums

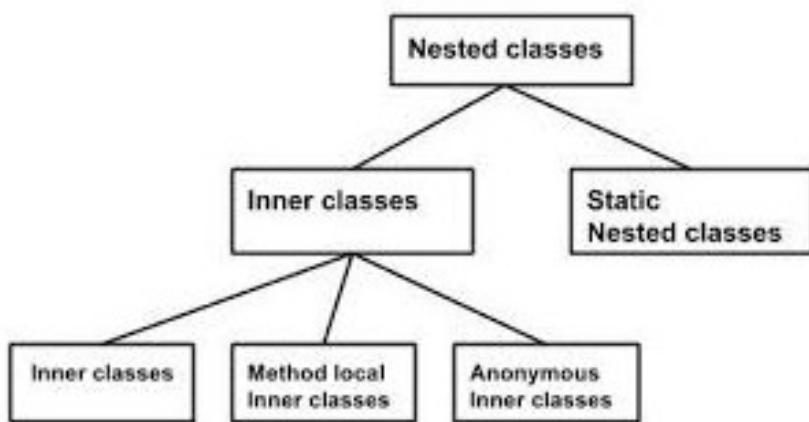
- Enum is a special type of classes.
- enum type used to put restriction on the instance values

```
public enum myCars{  
    HONDA,BMW,TOYOTA  
};  
.....  
.....  
myCars currentcar=myCars.BMW;  
System.out.println("My Current Cars : "+ myCars.BMW);
```

its optional !!!

```
enum Day {  
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY,  
    FRIDAY, SATURDAY  
}  
  
//used  
Day today = Day.WEDNESDAY;  
switch(today){  
    case SUNDAY:  
        break;  
    //...  
}
```

# Inner classes



- Inner class?
  - A class defined inside another class.
  - **Why to define inner classes**
- Inner classes
  - Top level inner class
  - Method local inner class
  - Anonymous Inner class
  - Method local argument inner class
- Static inner classes

# Top level inner class

- Non static inner class object can't be created without outer class instance
- All the private data of outer class is available to inner class
- Non static inner class can't have static members
  - <http://www.avajava.com/tutorials/lessons/iterator-pattern.html>

```
class A
{
    private int i=90;

    class B
    {
        int i=90;//  
        void foo()
        {
            System.out.println("instance value of outer class:"+ A.this.i);

            System.out.println("instance value of inner class:"+this.i);
        }
    }
    public class Inner1 {

        public static void main(String[] args) {
            A.B objectB=new A().new B();

            objectB.foo();
        }
    }
}
```

# Method local inner class

- Class define inside an method
- Can not access local data define inside method
- Declare local data final to access it

```
class A
{
    private int i=90;
    void foo()
    {
        int i=22;
        final int j=44;
        class B
        {
            void foofy()
            {
                //System.out.println("value of method local variable cant be access:"+i);
                System.out.println("value of method local" +
                    " can be accessed if it is declared final:"+ j);
            }
        }
        B b=new B();
        b.foofy();
    }
}
```

# Anonymous Inner class

- A way to implement polymorphism
- “On the fly”

```
interface Cookable
{
    public void cook();
}

class Food
{
    Cookable c=new Cookable() {
        @Override
        public void cook() {
            System.out.println("Cooked.....");
        }
    };
}
```

```
/* StaticClassDemo.java */

class OuterStatic
{
    private int mem = 20;
    private static int smem = 50;

    static class InnerStatic
    {
        public void accessMembers ()
        {
            System.out.println(mem); //Error: Cannot make a static reference to t
            System.out.println(smem);
        }
    }
}

public class StaticClassDemo
{
    public static void main(String[] args)
    {
        OuterStatic.InnerStatic is = new OuterStatic.InnerStatic();
        is.accessMembers();
    }
}
```

Remember that a static nested class cannot access non-static members of the outer class, because it does not have an implicit reference to any outer instance. Because a static class is not an inner class so it does not share any special relationship with an instance of the outer class.

# Object

- Object is an special class in java defined in `java.lang`
- Every class automatically inherit this class whether we say it or not...

We Writer like...

```
class Employee{  
    int id;  
    double salary;  
    ....  
    ....  
}
```

Java compiler convert it as...

```
class Employee extends Object{  
    int id;  
    double salary;  
    ....  
    ....  
}
```

- Why Java has provided this class?

# Method defined in Object class...

- String `toString()`
- boolean `equals()`
- int `hashCode()`
- `clone()`
- void `finalize()`
- `getClass()`
- Method that can't be overridden
  - final void `notify()`
  - final void `notifyAll()`
  - final void `wait()`

# toString( )

- If we do not override `toString()` method of `Object` class it print Object Identification number by default
- We can override it to print some useful information....

```
class Employee{  
    private int id;  
    private double salary;  
  
    public Employee(int id, double salary) {  
        this.id = id;  
        this.salary = salary;  
    }  
  
    .....  
    .....  
    Employee e=new Employee(22, 333333.5);  
    System.out.println(e);  
    .....  
    ....
```

**O/P**

com.Employee@addbf1

Java simply print object identification number not so useful message for client

# toString()

```
class Employee{  
    private int id;  
    private double salary;  
  
    public Employee(int id, double salary) {  
        this.id = id;  
        this.salary = salary;  
    }  
  
    @Override  
    public String toString() {  
        return "Employee [id=" + id + ", salary=" + salary + "]";  
    }  
}  
public class DemoToString {  
    public static void main(String[] args) {  
        Employee e=new Employee(22, 333333.5);  
        System.out.println(e);  
    }  
}
```

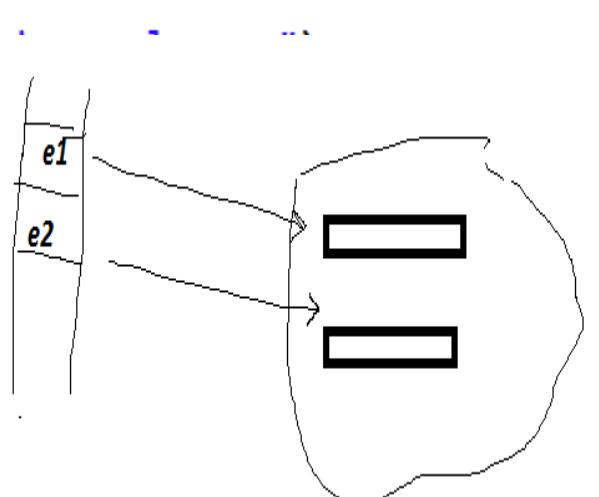
# equals

- - 
  - 
  - What O/P do you expect in this case.....

```
Employee e1=new Employee(22, 33333.5);
Employee e2=new Employee(22, 33333.5);

if(e1==e2)
    System.out.println("two employees are equals....");
else
    System.out.println("two employees are      ");
```

- O/P would be two employees are not equals.... ???
  - Problem is that using == java compare object id of two object and that can never be equals, so we are getting meaningless result...



# Overriding equals()

- Don't forget DRY run.....

```
@Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Employee other = (Employee) obj;
        if (id != other.id)
            return false;
        if (Double.doubleToLongBits(salary) != Double
                .doubleToLongBits(other.salary))
            return false;
        return true;
    }
```

# hashCode()



- 
- Whenever you override equals() for an type don't forget to override hashCode() method...
- hashCode() make DS efficient
- What hashCode does
  - HashCode divide data into buckets
  - Equals search data from that bucket...

```
@Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + id;
        long temp;
        temp = Double.doubleToLongBits(salary);
        result = prime * result + (int) (temp ^ (temp >>> 32));
        return result;
    }
```

# clone()



- Lets consider an object that creation is very complicated, what we can do we can make an clone of that object and use that
- Costly , avoid using cloning if possible, internally depends on serialization
- Must make class supporting cloning by implementing an marker interface ie Cloneable

```
class Employee implements Cloneable{  
    private int id;  
    private double salary;  
  
    public Employee(int id, double salary) {  
        this.id = id;  
        this.salary = salary;  
    }  
  
    @Override  
    protected Object clone() throws CloneNotSupportedException {  
        // TODO Auto-generated method stub  
        return super.clone();  
        //can write more code  
    }  
}
```

# finalize()

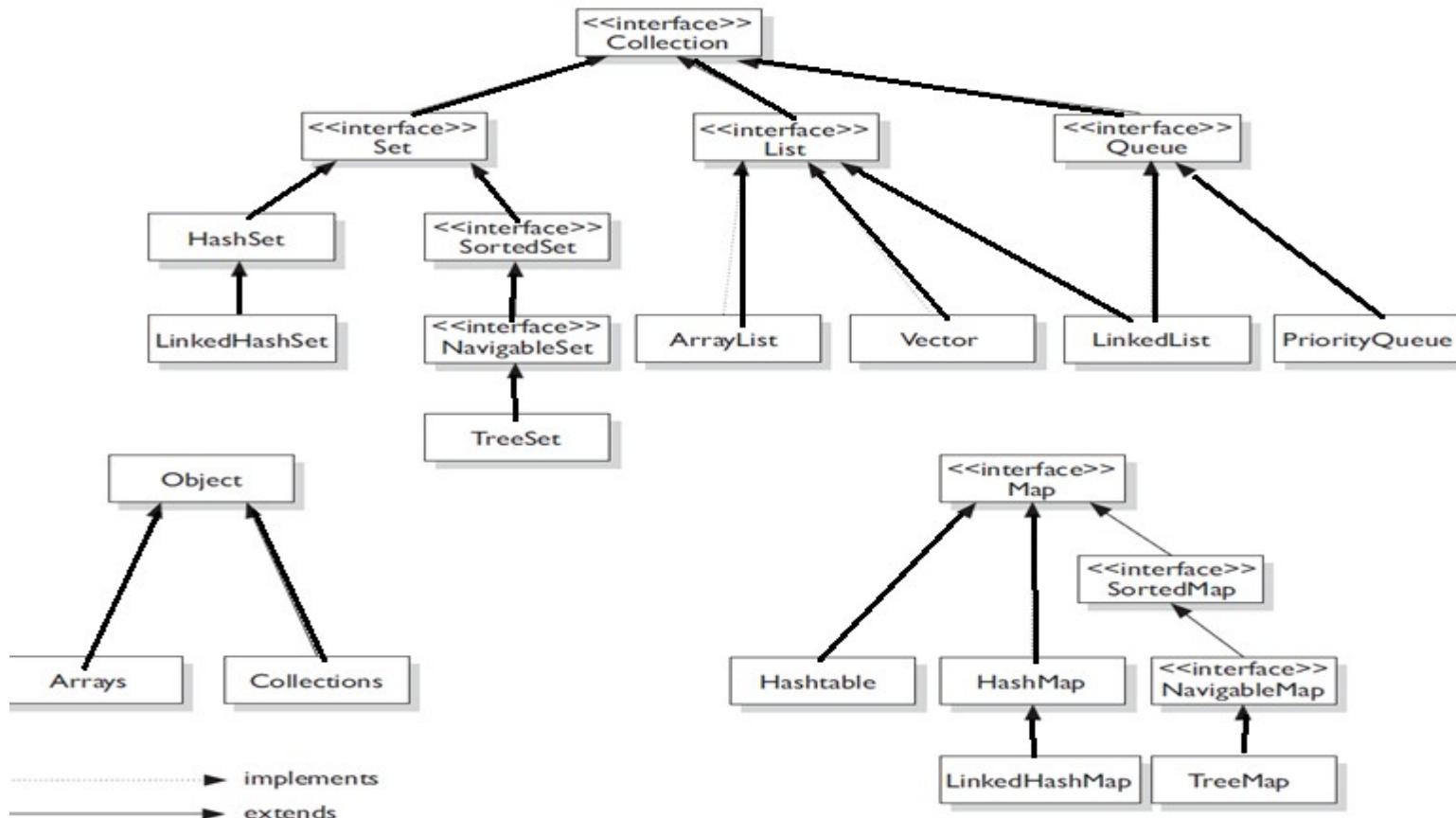
- As you are aware ..Java don't support destructor
- Programmer is free from memory management
- Memory mgt is done by an component of JVM ie called Garbage collector GC
- GC runs as low priority thread..
- We can override finalize() to request java
  - “Please run this code before recycling this object”
- Cleanup code can be written in finalize() method
- Not reliable, better not to use...
- **Demo programm**
  - WAP to count total number of employee object in the memory at any moment of time if an object is nullified then reduce count....

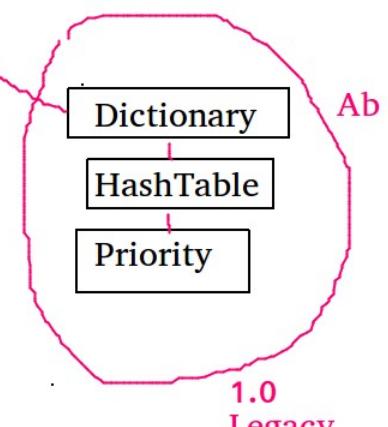
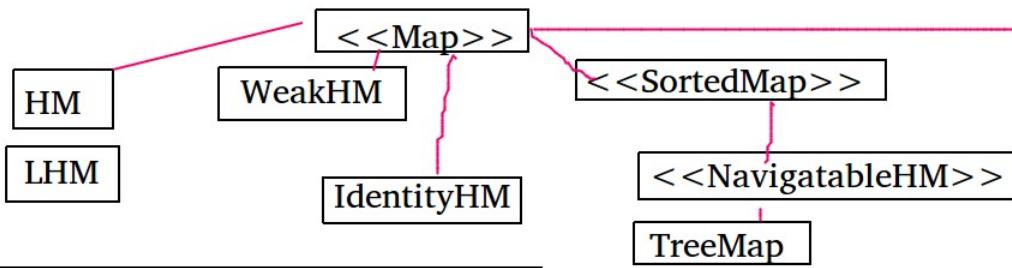
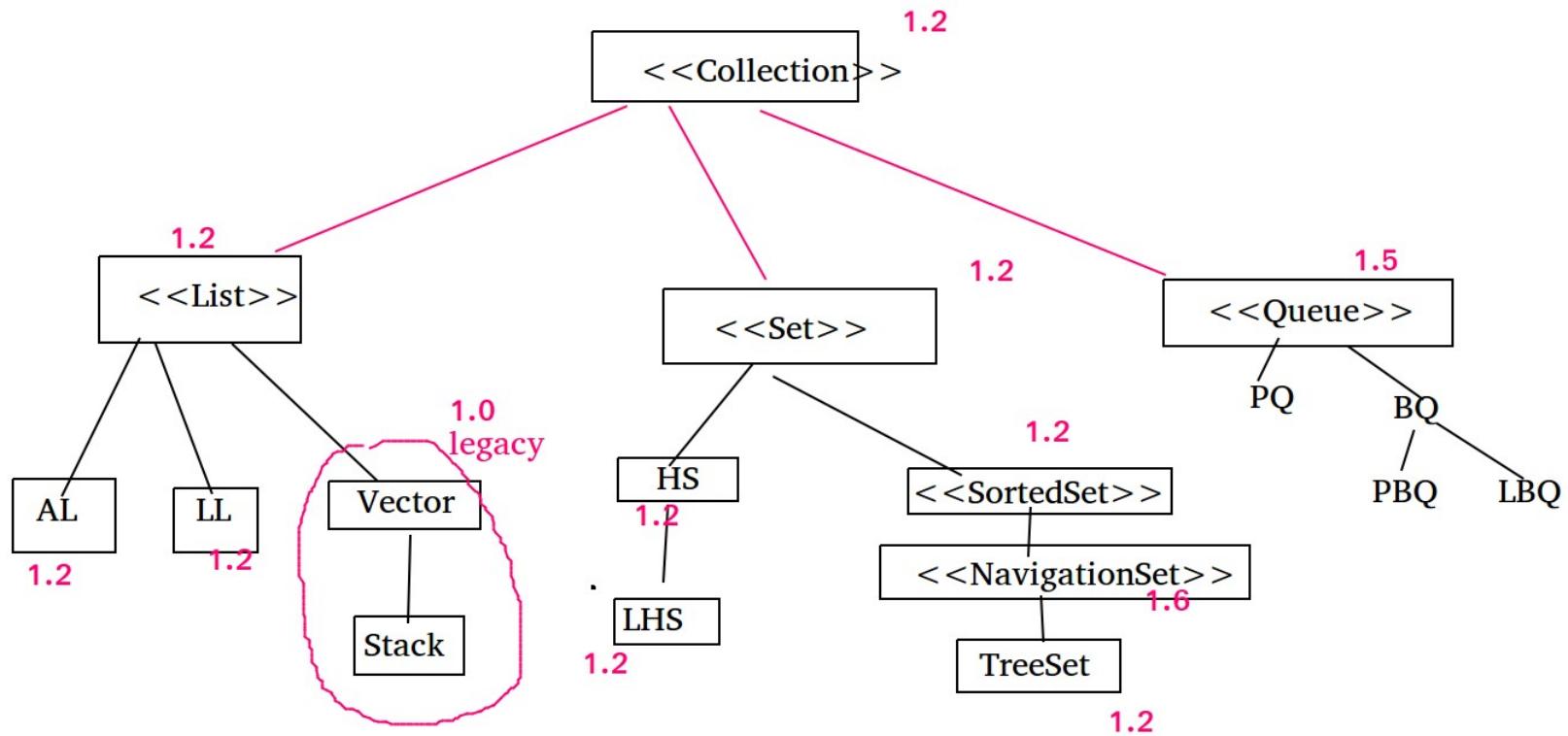
# Java collection

- Java collections can be considered a kind of readymade data structure, we should only need to know how to use them and how they work....
  - collection
    - Name of topic
  - Collection
    - Base interface
  - Collections
    - Static utility class provide various useful algorithm

# collection

- collection
  - Why it is called an framework?
  - Readymade Data structure in Java...





**Sorting:** Comparable/Comparator  
**Cursors:** Enumeration, Iterator, ListIterator  
**Utility classes:** Arrays, Collections

# Four type of collections

Collections come in four basic flavors:

- Lists Lists of things (classes that implement List).
- Sets Unique things (classes that implement Set).
- Maps Things with a *unique* ID (classes that implement Map).
- Queues Things arranged by the order in which they are to be processed.

# ArrayList: aka growable

```
List<String>list=new ArrayList<String>();  
...  
...  
list.size();  
list.contains("raj");  
  
test.remove("hi");  
  
Collections.sort(list);
```

## Note:

Collections.sort(list,Collections.reverseOrder());

Collections.addAll(list2,list1);

Add all elements from list1 to end of list2

Collections.frequency(list2,"foo");

print frequency of "foo" in the list2 collection

boolean flag=Collections.disjoint(list1,list);

return "true" if nothing is common in list1 and list2

Sorting with the Arrays Class

Arrays.sort(arrayToSort)

Arrays.sort(arrayToSort, Comparator)

# ArrayList of user defined object

```
class Employee{  
    int id;  
    float salary;  
    //getter setter  
    //const  
    //toString  
}  
  
List<Employee>list=new ArrayList<Employee>();  
  
list.add(new Employee(121,"rama"));  
list.add(new Employee(121,"rama"));  
list.add(new Employee(121,"rama"));  
  
System.out.println(list);  
  
Collections.sort(list);
```

How java can decide how  
to sort?

# Comparable and Comparator interface

- We need to teach Java how to sort user define object
- Comparable and Comparator interface help us to tell java how to sort user define object....

| Comparable                                | Comparator                                      |
|-------------------------------------------|-------------------------------------------------|
| java.lang                                 | java.util                                       |
| Natural sort                              | secondary sorts                                 |
| Only one sort sequence<br>is possible     | as many as you want                             |
| need to change the<br>design of the class | Dont need to change<br>desing of the class      |
| need to override                          | need to override                                |
| public int<br>compareTo(Employee o)       | public int<br>compare(Employee o1, Employee o2) |

# Implementing Comparable

```
class Employee implements Comparable<Employee>{
    private int id;
    private double salary;
    .....
    .....
    .....

    @Override
    public int compareTo(Employee o) {
        // TODO Auto-generated method stub
        Integer id1=this.getId();
        Integer id2=o.getId();
        return id1.compareTo(id2);
    }
}
```

# Comparator

- Don't need to change Employee class

```
class SalarySorter implements Comparator<Employee>{  
  
    @Override  
    public int compare(Employee o1, Employee o2) {  
        // TODO Auto-generated method stub  
        Double sal1=o1.getSalary();  
        Double sal2=o2.getSalary();  
  
        return sal1.compareTo(sal2);  
    }  
  
}
```

# Useful stuff

## Converting Arrays to Lists

---

```
String[] sa = {"one", "two", "three", "four"};
List sList = Arrays.asList(sa);
```

## Converting Lists to Arrays

---

```
List<Integer> iL = new ArrayList<Integer>();

for(int x=0; x<3; x++)
iL.add(x);

Object[] oa = iL.toArray(); // create an Object array

Integer[] ia2 = new Integer[3];

ia2 = iL.toArray(ia2); // create an Integer array
```

---

**Arrays.binarySearch(arrayFromWhichToSearch,"to search"))**

---

**return -ve no if no found**

**array must be sorted before hand otherwise o/p is not predictiale**

# Useful examples...

**user define funtion to print the arraylist/linkedlist**

```
printMe(list1);
...
...

public void printMe(List<String> list){
    for(String s:list)
    {
        System.out.println(s);
    }
}
```

**user define funtion to remove stuff from a arraylist /linkedlist**

```
removeStuff(list,2,5);
...
...

public void removeStuff(List<String> l, int from, int to)
{
    l.subList(from,to).clear();
}
```

# Useful examples...

## Merging two link lists

```
ListIterator ita=a.listIterator();
Iteratory itb=b.iterator();

while(itb.hasNext())
{
    if(ita.hasNext())
        ita.next();

    ita.add(itb.next());
}
```

## Removing every second element from an linkedList

```
itb=b.iterator();

while(itb.hasNext())
{
    itb.next();

    if(itb.hasNext())
    {
        itb.next();

        itb.remove();
    }
}
```

# LinkedList : AKA Doubly Link list..... can move back and forth.....

## Imp methods

---

```
boolean hasNext()  
Object next()  
boolean hasPrevious()  
Object previous()
```

## More methods

---

```
void addFirst(Object o);  
  
void addLast(Object o);  
  
Object getFirst();  
  
Object getLast();  
  
add(int pos, Object o);
```

# Useful examples...

**user define funtion to print linkedlist in reverse order**

```
reversePrint(list);
...
...

public void reversePrint(list<String>l ){

    ListIterator<String>it=l.iterator(l.size());

    while(it.hasPrevious())
        Sysout(it.previous());
}
```

# fundamental diff bw ArrayList and LinkedList

- ArrayLists manage arrays internally.  
[0][1][2][3][4][5] ....
- `List<Integer> arrayList = new ArrayList<Integer>();`
- LinkedLists consists of elements where each element has a reference to the previous and next element  
[0]->[1]->[2] ....  
  <-   <-

# ArrayList vs LinkedList

- Java implements ArrayList as array internally
  - Hence good to provide starting size
    - i.e. `List<String> s=new ArrayList<String>(20);` is better than `List<String> s=new ArrayList<String>();`
- Removing element from starting of arraylist is very slow?
  - `list.remove(0);`
  - if u remove first element, java internally copy all the element (shift by one)
- Adding element at middle in ArrayList is very inefficient...

# Performance ArrayList vs LinkedList !!!

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class App
{
    public static void main(String[] args)
    {
        List<Integer> arrayList = new ArrayList<Integer>();
        List<Integer> linkedList = new LinkedList<Integer>();

        doTimings("ArrayList", arrayList);
        doTimings("LinkedList", linkedList);
    }
}
```

```
private static void doTimings(String type, List<Integer> list)
{
    for(int i=0; i<1E5; i++)
        list.add(i);

    long start = System.currentTimeMillis();

    /*
     * Add items at end of list
     */
    for(int i=0; i<1E5; i++)
    {
        list.add(i);
    }

    /*
     * Add items elsewhere in list
     */
    for(int i=0; i<1E5; i++)
    {
        list.add(0, i);
    }

    long end = System.currentTimeMillis();

    System.out.println("Time taken: " + (end - start) + " ms for " + type);
}
```

```
Time taken: 7546 ms for ArrayList
Time taken: 76 ms for LinkedList
```

# HashMap

- Key ---->Value
- declaring an hashmap
  - `HashMap<Integer, String> map = new HashMap<Integer, String>();`
- Populating values
  - `map.put(5, "Five");`
  - `map.put(8, "Eight");`
  - `map.put(6, "Six");`
  - `map.put(4, "Four");`
  - `map.put(2, "Two");`
- Getting value
  - `String text = map.get(6);`
  - `System.out.println(text);`

# Looping through HashMap

```
for(Integer key: map.keySet())
{
    String value = map.get(key);

    System.out.println(key + ":" + value);
}
```

most imp thing to remember

-----  
order of getting key value is not maintained

ie hashMap dont keep key and value in any particular order

# Other map variants

- LinkedHashMap
  - Aka. Doubly link list
  - key and value are in same order in which you have inserted.....
- TreeMap
  - sort keys in natural order(what is natural order?)
  - for int
    - 1,2,3.....
  - for string
    - "a","b".....
  - For user define key
    - Define sorting order using Comparable /Comparator
-

# set

- Don't allow duplicate element

three types:

-----  
hashset  
linkedhashset  
treeset

HashSet does not retain order.

-----  
`Set<String> set1 = new HashSet<String>();`

LinkedHashSet remembers the order you added items in

-----  
`Set<String> set1 = new LinkedHashSet<String>();`

TreeSet sorts in natural order

-----  
`Set<String> set1 = new TreeSet<String>();`

Printing freq of unique words  
from a file in increasing order of freq

| words | freq |
|-------|------|
| Apple | 7    |
| Ball  | 5    |
| ...   |      |

# User define key in HashMap

- If you are using user define key in HashMap do not forget to override hashCode for that class
- Why?
  - We may not find that content again !

# HashMap vs Hashtable

- Hashtable is threadsafe, slow as compared to HashMap
- Better to use HashMap
  
- Some more interesting difference
  - Hashtable give runtime exception if key is “null” while HashMap don’t

# HashMap Internal Structure :

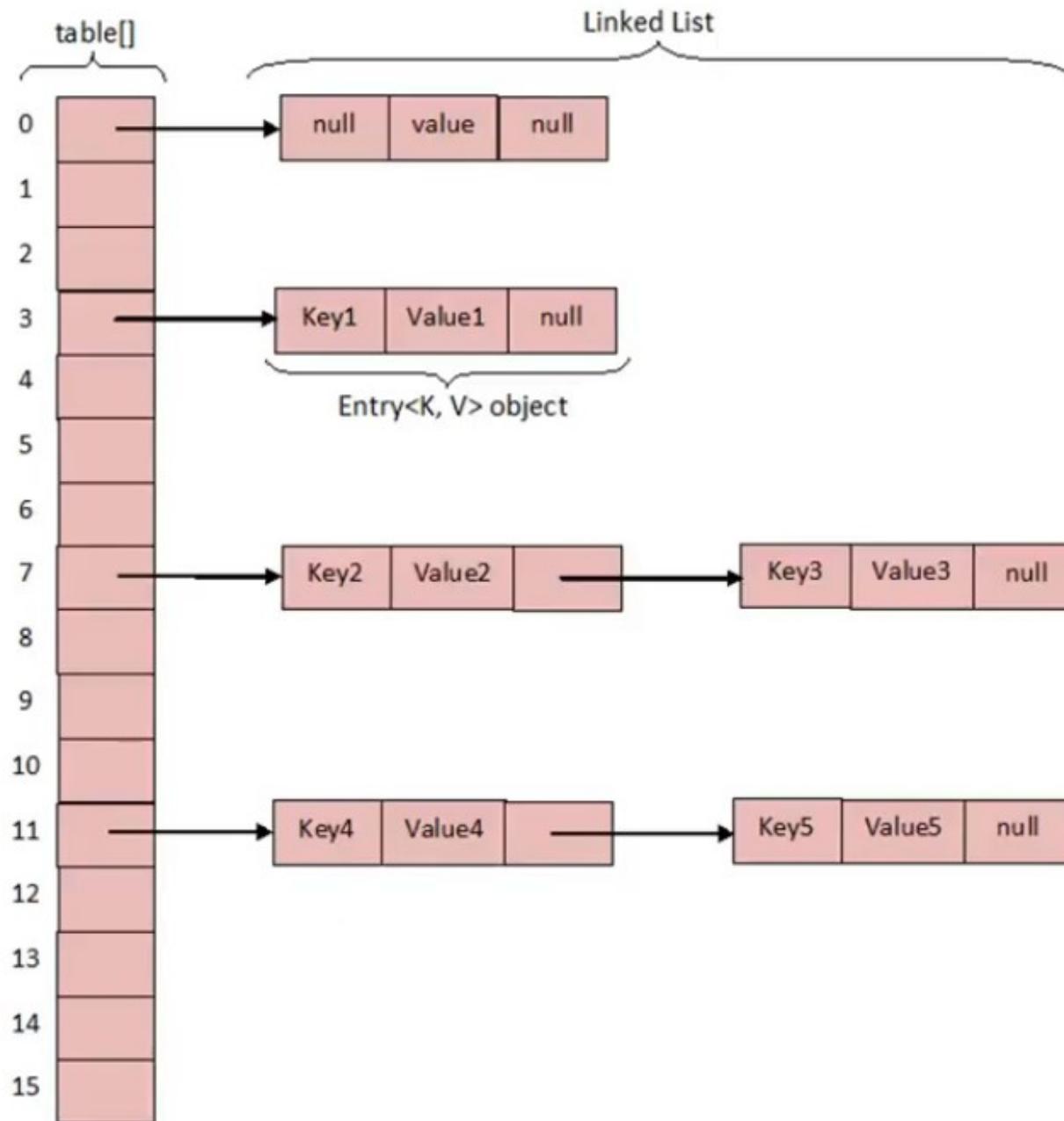
- ▶ HashMap stores the data in the form of key-value pairs.
- ▶ Each key-value pair is stored in an object of Entry<K, V> class.

```
static class Entry<K,V> implements Map.Entry<K,V>
{
    final K key;
    V value;
    Entry<K,V> next;
    int hash;

}
```

- ▶ **key** : It stores the key of an element and its final.
- ▶ **value** : It holds the value of an element.
- ▶ **next** : It holds the pointer to next key-value pair. This attribute makes the key-value pairs stored as a linked list.
- ▶ **hash** : It holds the hashCode of the key.

## HashMap Internal Structure



```
Map score=new HashMap<String,Integer>();
```

```
Score.put("Kohli",100)
```

```
Hash(kohli) -> 45612
```

```
indexFor(hash code) -> 3
```

```
Score.put("Sachin",200)
```

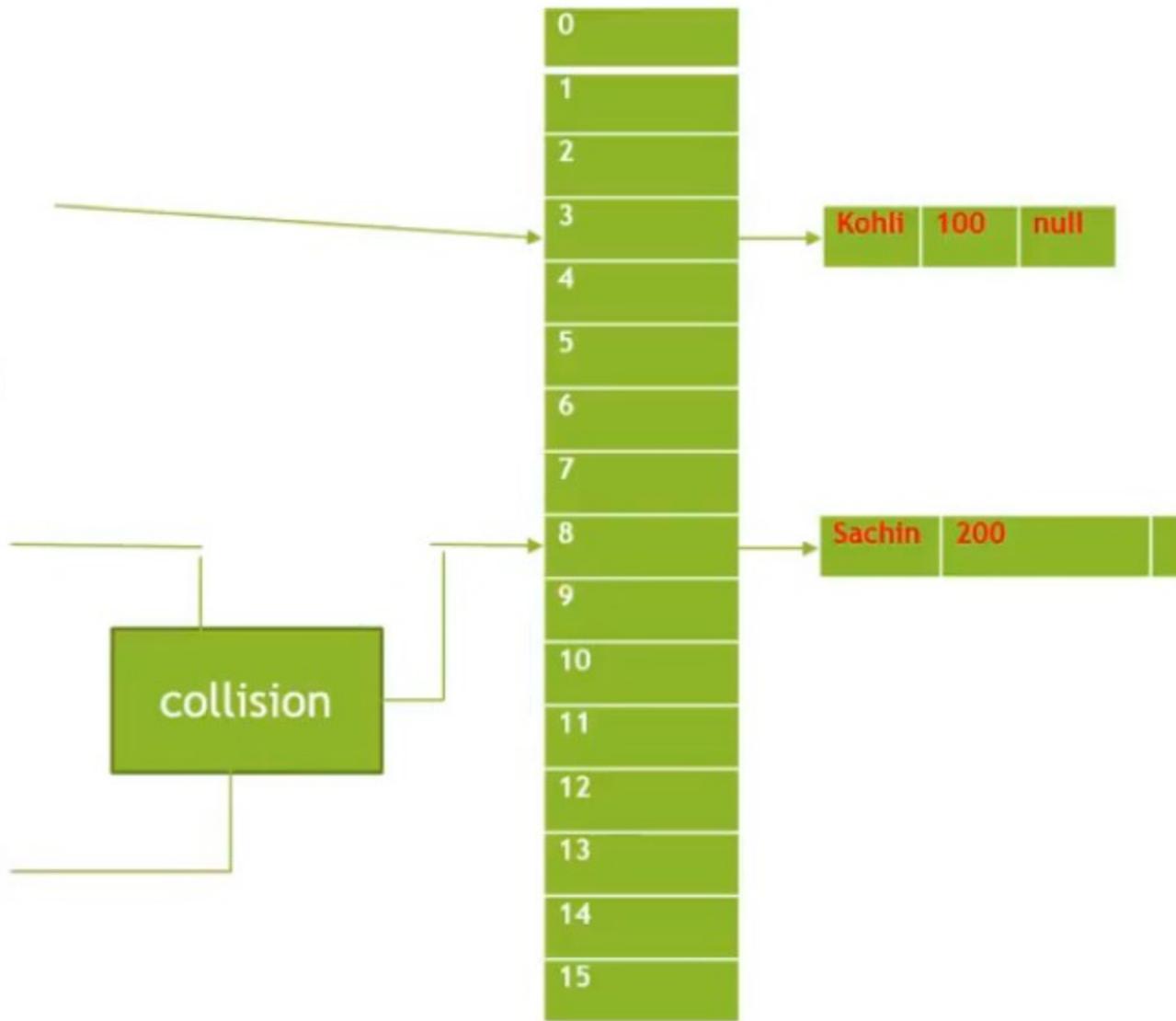
```
Hash("Sachin") -> 2000
```

```
indexFor(hashcode) -> 8
```

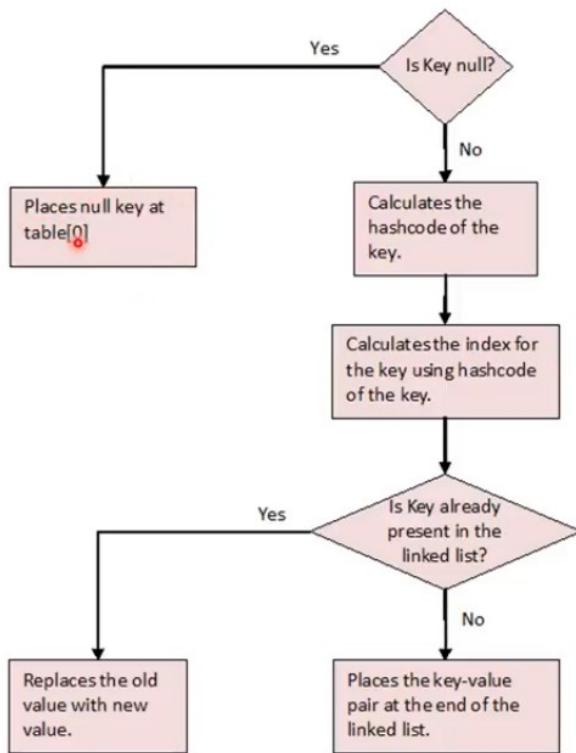
```
Score.put("Dhoni",300)
```

```
Hash("Dhoni") -> 2000
```

```
indexFor(hashcode) -> 8
```



### Flowchart Of put() Method



- ▶ In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.
- ▶ If key given already exist in HashMap, the value is replaced with new value.
- ▶ hash code of null key is 0.
- ▶ When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

## How get() method Works?

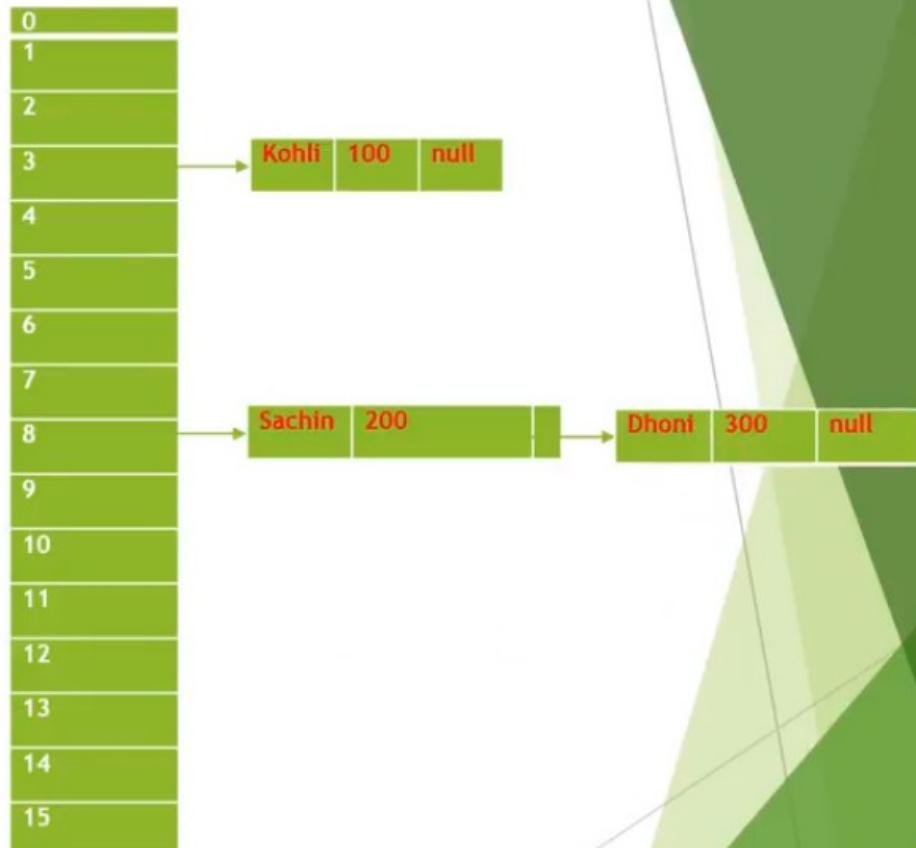
Score.get("Dhoni")

Calculate hash code of Key {"Dhoni"}. It will be generated as 2000.

Calculate index by using index method it will be 8.

Go to index 8 of array and compare first element's key with given key. If both are equals then return the value, otherwise check for next element if it exists.

In our case it is found as second element and returned value is 300.



# Generics

## □ Before Java 1.5

- `List list=new ArrayList();`
  - Can add anything in that list
  - Problem while retrieving

## □ Now Java 1.5 onward

- `List<String> list=new ArrayList<String>();`  
`list.add("foo");//ok`  
`list.add(22);// compile time error`
- Generics provide type safety
- Generics is compile time phenomena...

# Issues with Generics

- Try not to mix non Generics code and Generics code...we can have strange behaviour.

```
package com;
import java.util.*;

public class DemoGen1 {
    public static void main(String[] args) {

        List<String> list=new ArrayList<String>();
        list.add("foo");
        list.add("bar");

        strangMethod(list);

        for(String temp:list)
            System.out.println(temp);
    }

    private static void strangMethod(List list) {
        list.add(new Integer(22)); // OMG.....
    }
}
```

# Polymorphic behaviour

```
class Animal {  
}  
  
class Cat extends Animal{  
}  
  
class Dog extends Animal{  
}
```

```
Animal []aa=new Cat[4] ;// allowed
```

```
List<Animal>list=new ArrayList<Cat>()
```



# <? extends XXXXX> aka upper bound

```
package com;
import java.util.*;

public class DemoGen1 {
    public static void main(String[] args) {

        List<Integer> list=new ArrayList<Integer>();
        list.add(22);
        list.add(33);

        strangMethod(list);

        for(Integer temp:list)
            System.out.println(temp);
    }

    private static void strangMethod(List<? extends Number> list) {
        list.add(new Integer(22)); //Compile time error..... Good
    }
}
```

in strangMethod() we can pass any derivative of Number class but we are not allowed to modify the list

Upper-bound is when you specify (? extends Field) means argument can be any Field or **subclass** of Field.  
Lower-bound is when you specify (? super Field) means argument can be any Field or **superclass** of Field.

# <? Super XXXX> aka lower bound

```
class Animal {  
}  
  
class Cat extends Animal{  
}  
  
class Dog extends Animal{  
}  
  
class CostlyDog extends Dog{  
}  
.....  
.....  
List<Dog>list=new ArrayList<Dog>();  
list.add(new Dog("white"));  
list.add(new Dog("red"));  
list.add(new Dog("black"));  
strangMethod(list);  
  
private static void strangMethod(List<? super Dog> list) {  
    list.add(new CostlyDog());  
}  
.....  
.....
```

Anytype of Dog is allowed and can also modify list

# Generic class

```
class MyObject<T>{
    T myObject;

    public T getMyObject() {
        return myObject;
    }

    public void setMyObject(T myObject) {
        this.myObject = myObject;
    }

}
public class GenClass {

    public static void main(String[] args) {
        MyObject<String> o=new MyObject<String>();
        o.setMyObject(new Integer(22));
        //System.out.println(it.intValue());
    }
}
```

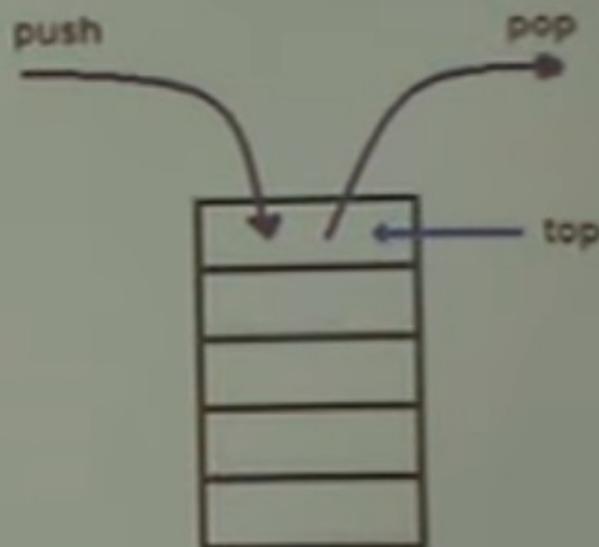
*will not compile !!!*

# Generic method

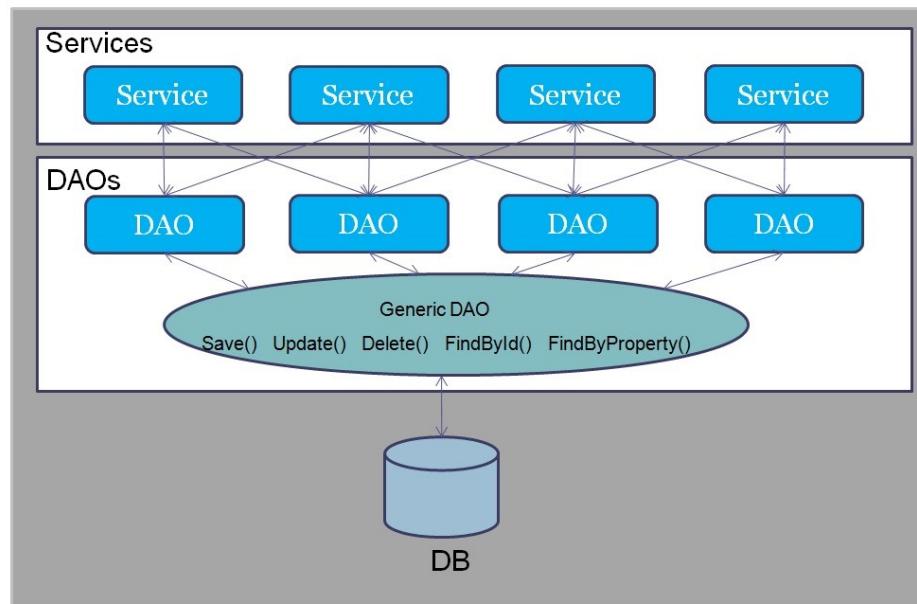
```
class MaxOfThree{  
  
    public static <T extends Comparable<T>> T maxi(T a,T b, T c){  
        T max=a;  
        if(b.compareTo(a)>0)  
            max=b;  
        if(c.compareTo(max)>0)  
            max=c;  
        return max;  
    }  
}
```

# Generics – Stack (of Objects) Class

```
class Stack<E>
{
    private final int size;
    private int top;
    private E[] elements;
    public Stack() { this(10); }
    public Stack(int s) {
        size = s > 0 ? s : 10;
        top = -1;
        elements = (E[]) new Object[size]; // create array
    }
    public void push(E pushValue) {
        if (top < size - 1) // if stack is not full
            elements[++top] = pushValue; // place pushValue on Stack
    }
    public E pop() {
        if (top > -1) // if stack is not empty
            return elements[top--]; // remove and return top element of Stack
    }
}
```



# Generic Dao





# Java Reflection

## Java Reflection:

- => Reflection is aka class manipulator?
  - => Used to manipulate classes and everything in a class
  - => Can slow down a program because the JVM can not optimize the code
  - => Can not used with applets
  - => Should be used sparingly
- What is Java reflection?

"Reflection is the process of analysing the capabilities of a class at runtime"

That is analysing the details about the class which include the methods in the class, the variables of the class, the constructors in the class, the interface that the class is implementing, the methods that are coming from the interface

To gather all the above details is provided by reflection api

Reflection api is not the project development but used for PRODUCT development

Example application area:

JVM development, server design, framework design, tool design, compiler

# Java Reflection

```
Private class A {  
}  
}
```



**Comment:** An outer class cannot have a PRIVATE declaration. An inner class can have PRIVATE declaration. A class can be declared as PUBLIC and DEFAULT; it cannot HAVE PRIVATE or PROTECTED access modifier

## COMPILER:

The compiler will read this class according to the CLASS RULES defined in the compiler.

## CLASS RULES

Can have only PUBLIC and DEFAULT access modifier in outer class. Will not allow PRIVATE OR PROTECTED access modifier for outer class. This rule is analysed by the REFLECTION API present in the compiler and it shows an error “MODIFIER PRIVATE not allowed”

Similarly rules for methods in the class, constructors in the class, interfaces inherited by the class, super classes extended by the class are also present and their access modifiers are also checked by REFLECTION API present in the compiler

# Java Reflection

## Reflection Package

`java.lang.reflect` package provides classes for performing reflection api.

Some classes in this reflect packages are :

`java.lang.Class` :

base class to perform reflection api.  
This is used together to get metadata information about class

`java.lang.reflect.Field`:

Complete declarative information of a particular variable ie.  
metadata of a particular variable like access modifier, datatype,  
value and name of variable etc

`java.lang.reflect.Method`:

complete declarative information about a method ie. is able to store metadata  
about methods, method names, access modifier, return type, parameters in method,  
exception thrown etc

`java.lang.reflect.Constructor`:

information about constructs like name of constructors, access modifier, parameter type

`java.lang.reflect.Modifier`:

complete metadata about access modifier

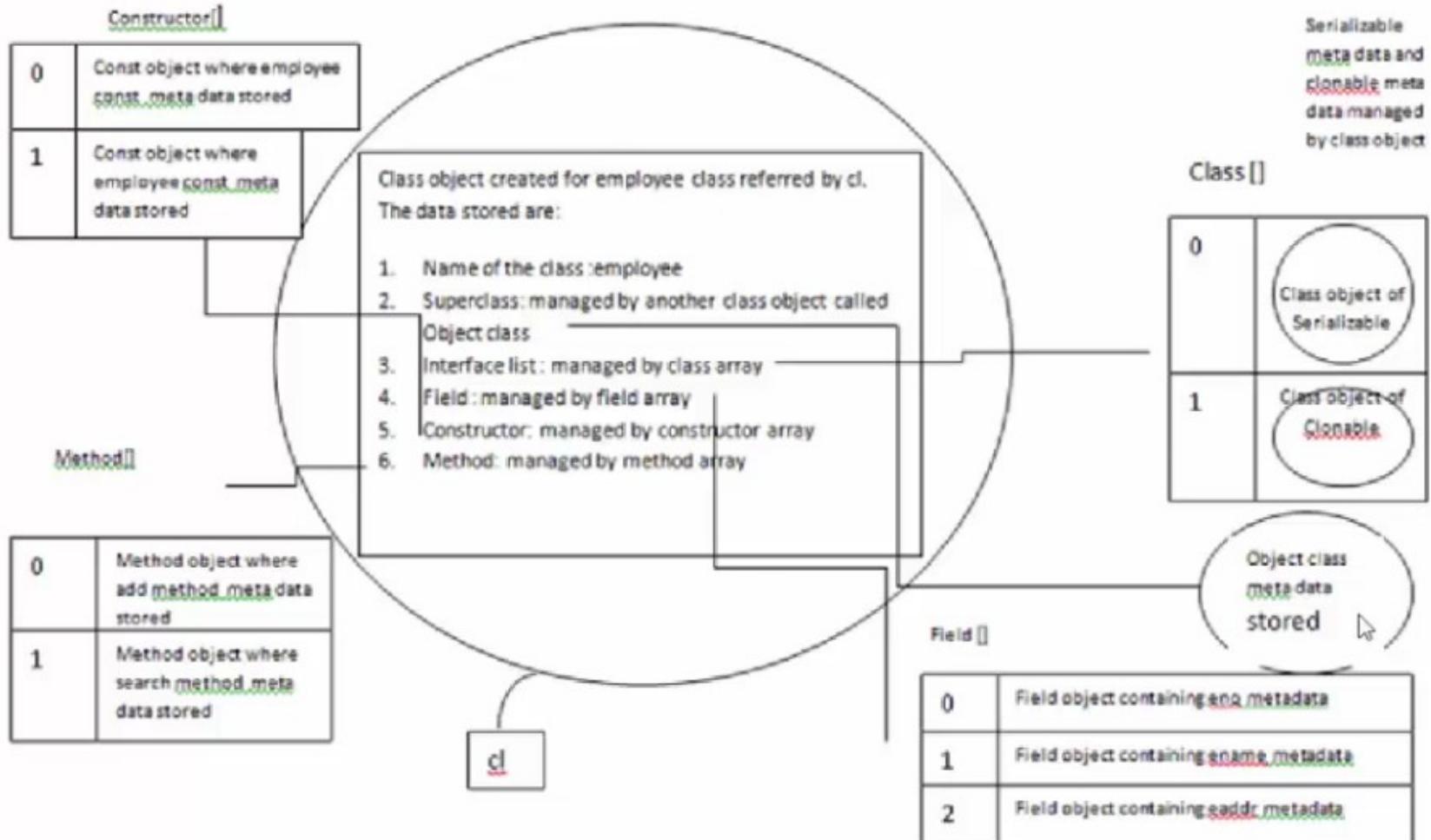
# Java Reflection

```
public class Employee implements Serializable, Cloneable{
    // variable ==4
    public int empNo;
    public static int companyName;
    public String name;
    public final String add="delhi";

    // ctr==2
    public Employee(int empNo, String name) {
        this.empNo = empNo;
        this.name = name;
    }
    public Employee() {}

    // methods==2
    public void add(int id, String name, String address){
    }
    public Employee search(int id){return new Employee();}
}
```

# Java Reflection



# Changing behaviour at runtime

Anyway, reflection does not allow you to change code behaviour, it can only explore current code, invoke methods and constructors, change fields values, that kind of things.

If you want to actually change the behaviour of a method you would have to use a bytecode manipulation library such as ASM. But this will not be very easy, probably not a good idea...

Patterns that might help you :

- If the class is not final and you can modify the clients, extend the existing class and overload the method, with your desired behaviour. Edit : that would work only if the method were not static !
- Aspect programming : add interceptors to the method using AspectJ

Anyway, the most logical thing to do would be to find a way to modify the existing class, work-arounds will just make your code more complicated and harder to maintain.

# Annotation

## Annotations starting from 5.0.

This feature was added to Java 5.0 as a result of the JSR 175 namely “A Metadata Facility for the JavaTM Programming Language”.

## Built-in Annotations in Java

@Override  
@Deprecated  
@SuppressWarnings  
@Target  
@Retention  
@FunctionalInterface

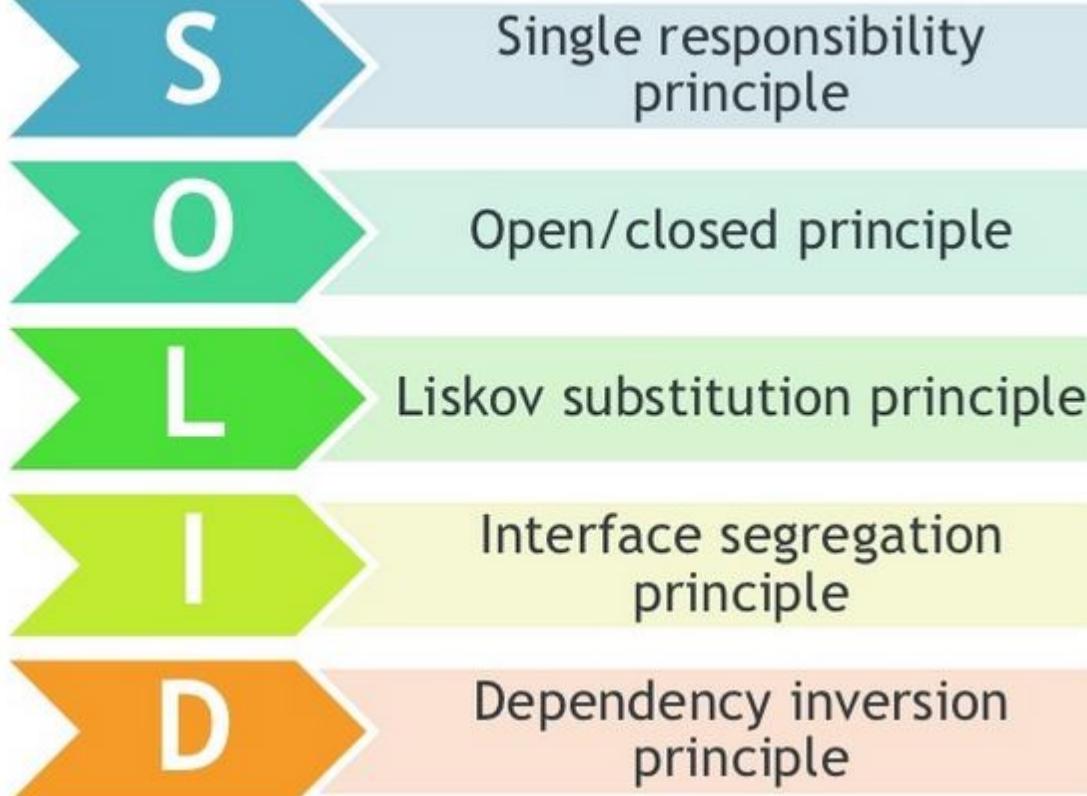
## User-defined Annotations

```
Target({ElementType.FIELD, ElementType.LOCAL_VARIABLE})  
public @interface Persistable{  
    String fileName() default "defaultMovies.txt";  
}
```

# Top 10 Object Oriented Design Principles

---

1. DRY (Don't repeat yourself) - avoids duplication in code.
2. Encapsulate what changes - hides implementation detail, helps in maintenance
3. Open Closed design principle - open for extension, closed for modification
4. SRP (Single Responsibility Principle) - one class should do one thing and do it well
5. DIP (Dependency Inversion Principle) - don't ask, let framework give to you
6. Favor Composition over Inheritance - Code reuse without cost of inflexibility
7. LSP (Liskov Substitution Principle) - Sub type must be substitutable for super type
8. ISP (Interface Segregation Principle) - Avoid monolithic interface, reduce pain on client side
9. Programming for Interface - Helps in maintenance, improves flexibility
10. Delegation principle - Don't do all things by yourself, delegate it



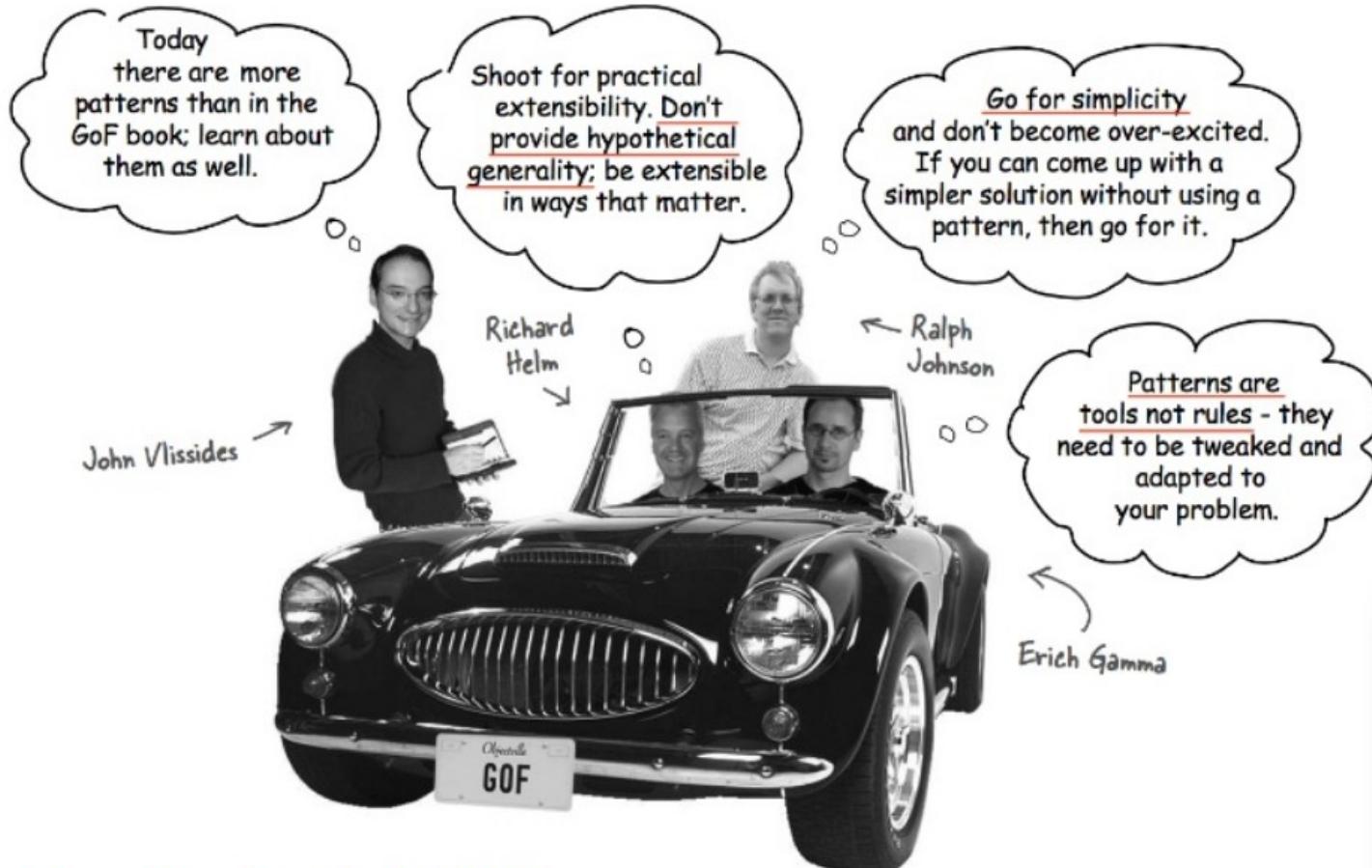
# Design pattern

- Proven way of doing things
- **Gang of 4 design patterns ???**
- **total 23 patterns**
- **Classification patterns**
  1. **Creational**
  2. **Structural**
  3. **Behavioral**

“In software engineering, a software design pattern is a general **reusable solution** to a commonly occurring problem within a **given context** in software design. It is not a **finished design** that can be transformed directly into source or machine code. It is a **description** or **template** for how to solve a problem that can be used in many different situations.”

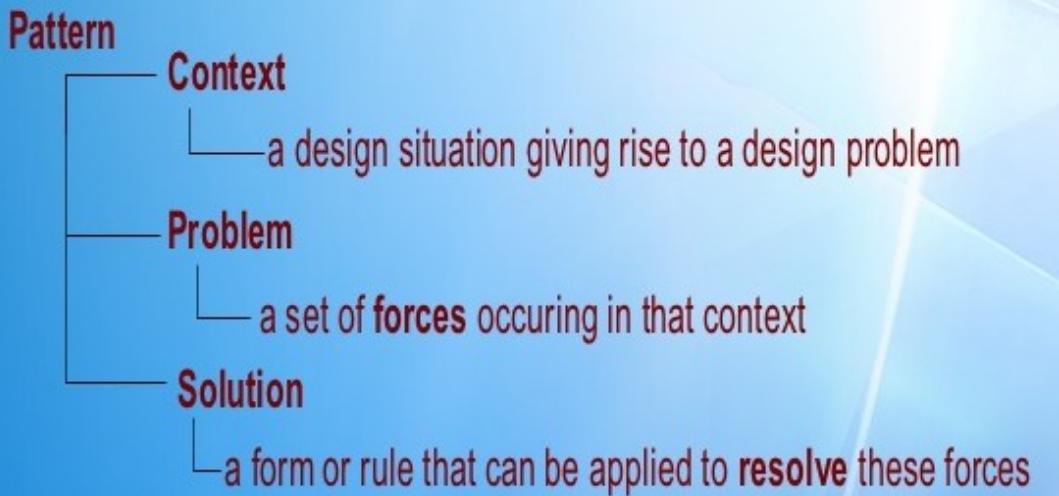
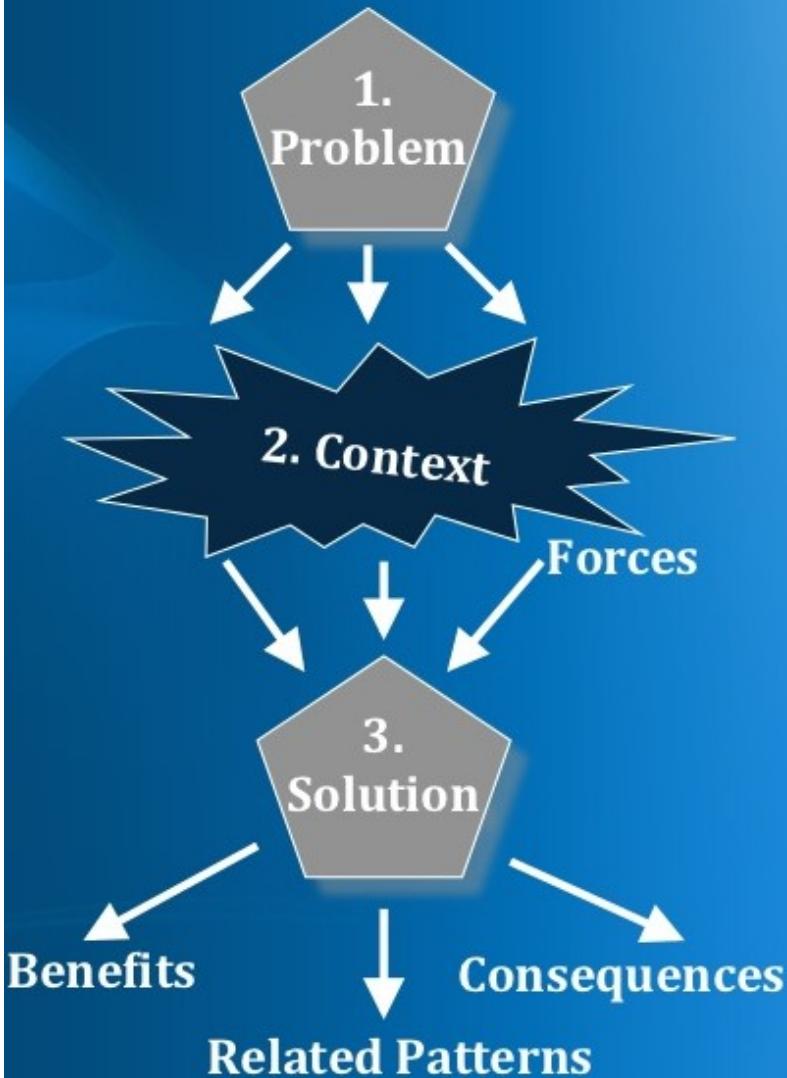


WIKIPEDIA  
The Free Encyclopedia



Keep it simple (KISS)





**IF** you find yourself in **CONTEXT**  
for example **EXAMPLES**,  
with **PROBLEM**,  
entailing **FORCES**  
**THEN** for some **REASONS**,  
apply **DESIGN FORM AND/OR RULE**  
to construct **SOLUTION**  
leading to **NEW CONTEXT & OTHER PATTERNS**

# DESIGN PATTERNS – CLASSIFICATION

## Structural Patterns

- 1. Decorator
- 2. Proxy
- 3. Bridge
- 4. Composite
- 5. Flyweight
- 6. Adapter
- 7. Facade

## Creational Patterns

- 1. Prototype
- 2. Factory Method
- 3. Singleton
- 4. Abstract Factory
- 5. Builder

## Behavioral Patterns

- 1. Strategy
- 2. State
- 3. TemplateMethod
- 4. Chain of Responsibility
- 5. Command
- 6. Iterator
- 7. Mediator
- 8. Observer
- 9. Visitor
- 10. Interpreter
- 11. Memento

day2

## Day 2:

- **Introduction to Stream programming, Why i should care for it?**
  - functional interface, examples of functional interface
  - @FunctionalInterface
  - Exploring existing functional interface in JDK
  - Methods vs. Functions, understanding immutability
  - Interface evolution, static method and default method inside interface
  - Issues with diamond problem interface java 8, rules to resolve it
- **What is lambda expression?**
  - Lambdas VS Anonymous Inner Classes
  - performance, how internally lambda expression works?
- **Passing code with behavior parameterization**
  - introduction to stream processing
  - Collections Streams and Filters
  - Iterate using forEach methods of Streams and List
  - Describe Stream interface and Stream pipeline
  - Filter a collection by using lambda expressions
  - Use method references with Streams
- **Lambda Built-in Functional Interfaces**
  - Use the built-in interfaces included in the java.util.function package such as
  - Predicate, Consumer, Function, and Supplier
  - Develop code that uses primitive versions of functional interfaces
  - Develop code that uses binary versions of functional interfaces
  - Develop code that uses the UnaryOperator interface

# Overview

JAVA 8 RELEASE (March 18, 2014)



Lambdas &  
Method references



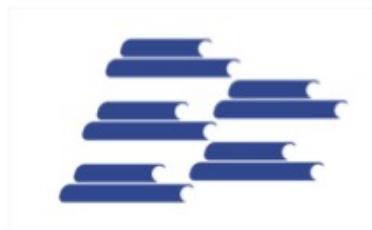
Optional (value wrapper)



METHODS INSIDE JAVA INTERFACES



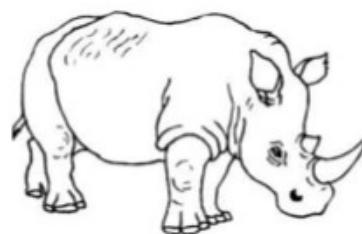
New Utilities



Stream API



Date Time API



Nashorn, JS Engine

# Java 8: Interfaces and Abstract Classes

|                  | Java 7 and Earlier                                                                                                                                                                                                                    | Java 8 and Later                                                                                                                                                                                                                       |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Abstract Classes | <ul style="list-style-type: none"><li>• Can have concrete methods and abstract methods</li><li>• Can have static methods</li><li>• Can have instance variables</li><li>• Class can directly extend one</li></ul>                      | (Same as Java 7)                                                                                                                                                                                                                       |
| Interfaces       | <ul style="list-style-type: none"><li>• Can only have abstract methods – no concrete methods</li><li>• Cannot have static methods</li><li>• Cannot have mutable instance variables</li><li>• Class can implement any number</li></ul> | <ul style="list-style-type: none"><li>• Can have concrete (default) methods and abstract methods</li><li>• Can have static methods</li><li>• Cannot have mutable instance variables</li><li>• Class can implement any number</li></ul> |

Conclusion: there is little reason to use abstract classes in Java 8. Except for instance variables, Java 8 interfaces can do everything that abstract classes can do, plus are more flexible since classes can implement more than one interface. This means (arguably) that Java 8 has real multiple inheritance.

# Static Methods in Interfaces

- **Idea**

- Java 7 and earlier prohibited static methods in interfaces. Java 8 now allows this

- **Motivation**

- Seems natural to put operations related to the general type in the interface
  - Does not violate the “spirit” of interfaces
    - `Shape.sumAreas(arrayOfShapes);`

- **Notes**

- You must use interface name in the method call, even from code within a class that implements the interface
    - `Shape.sumAreas`, not `sumAreas`
  - The static methods cannot manipulate static variables
    - Java 8 interfaces continue to prohibit mutable fields

# Shape

```
public interface Shape {  
    double getArea(); // All real shapes must define a getArea  
  
    public static double sumAreas(Shape[] shapes) {  
        double sum = 0;  
        for(Shape s: shapes) {  
            sum = sum + s.getArea();  
        }  
        return(sum);  
    }  
}
```

# Circle

```
public class Circle implements Shape {  
    private double radius;  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
  
    @Override  
    public double getArea() {  
        return(Math.PI * radius * radius);  
    }  
  
    ...  
}
```

Rectangle and Square are similar.

# ShapeTest

```
public class ShapeTest {  
    public static void main(String[] args) {  
        Shape[] shapes = { new Circle(10), // Area is about 314.159  
                           new Rectangle(5, 10), // Area is 50  
                           new Square(10) }; // Area is 100  
        System.out.println("Sum of areas: " +  
                           Shape.sumAreas(shapes));  
                           // Area is about 464.159  
    }  
}
```

# No Conflicts: Java 7

- **Interfaces Int1 and Int2 specify someMethod**

```
public interface Int1 { int someMethod(); }
public interface Int2 { int someMethod(); }
```

- **Class ParentClass defines someMethod**

```
public class ParentClass {
    public int someMethod() { return(3); }
}
```

- **Examples**

```
public class SomeClass implements Int1, Int2 { ... }
```

- No conflict: SomeClass must define someMethod, and by doing so, satisfies both interfaces

```
public class ChildClass extends ParentClass implements Int1 { ... }
```

- No conflict: the child class inherits someMethod from ParentClass, and interface is satisfied

# Potential Conflicts: Java 8

- **Interfaces Int1 and Int2 define someMethod**

```
public interface Int1 { default int someMethod() { return(5); } }
public interface Int2 { default int someMethod() { return(7); } }
```

- **Class ParentClass defines someMethod**

```
public class ParentClass {
    public int someMethod() { return(3); }
}
```

- **Examples**

```
public class SomeClass implements Int1, Int2 { ... }
```

- Potential conflict: whose definition of someMethod wins, the one from Int1 or the one from Int2?

```
public class ChildClass extends ParentClass implements Int1 { ... }
```

- Potential conflict: whose definition of someMethod wins, the one from ParentClass or the one from Int1?

# Resolving Conflicts

- **Classes win over interfaces**

```
public class ChildClass extends ParentClass implements Int1
```

- Conflict resolved: the version of someMethod from ParentClass wins over the version from Int1
- This rule also means that interfaces cannot provide default implementations for methods from Object (e.g., `toString`)
  - The methods from the interface could *never* be used, so Java prohibits you from even writing them

- **Conflicting interfaces: you must redefine**

```
public class SomeClass implements Int1, Int2
```

- The conflict cannot be resolved automatically, and SomeClass must give a new definition of someMethod
- But, this new method can refer to one of the existing methods with `Interface1.super.someMethod(...)` or `Interface2.super.someMethod`

# Summary

- **Static methods**

- Use for methods that apply *to* instances of that interface
  - Shape.sumAreas(Shape[] shapes)
  - Op.timeOp(Op opToTime)

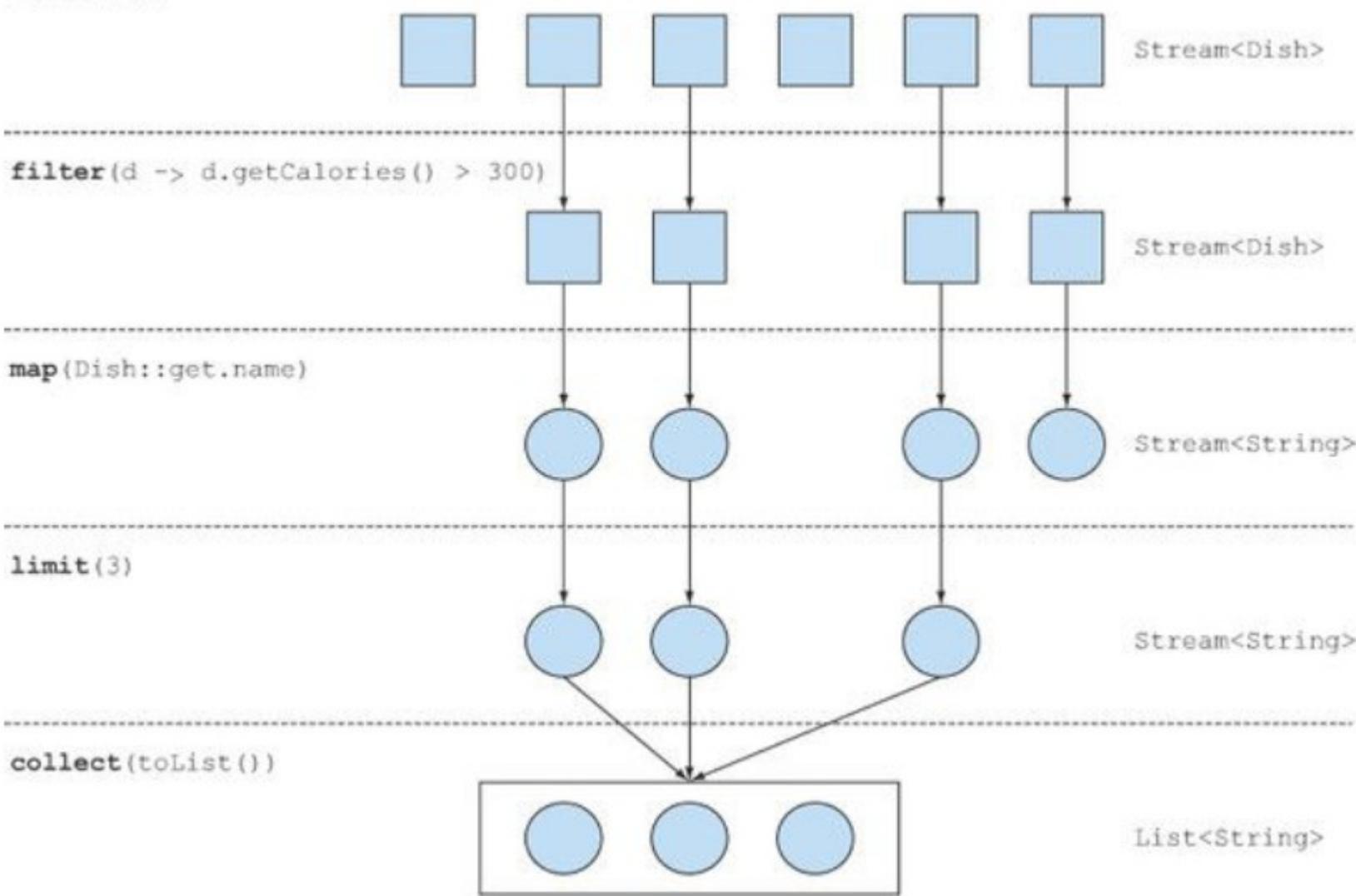
- **Default methods**

- Use to add behavior to existing interfaces without breaking classes that already implement the interface
- Use for operations that are called *on* instances of your interface type
- Resolving conflicts
  - Classes win over interfaces
  - If two interfaces conflict, class must reimplement the method
    - But the new method can refer to old method by using InterfaceName.super.methodName

# 1. example stream operation

Menu stream

Example stream operation



## 2. stream vs collection

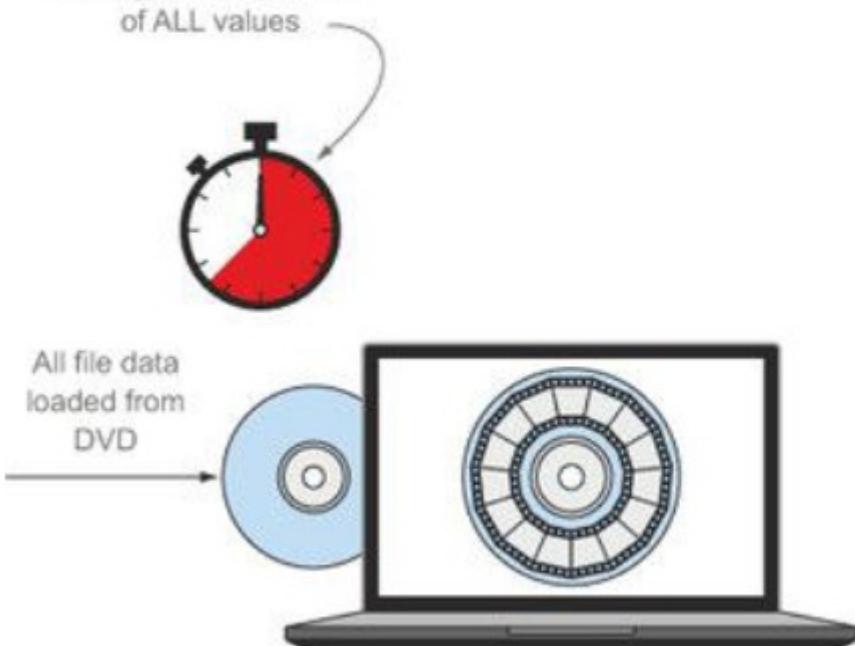
## Streams vs. collections

A collection in Java 8 is like  
a movie stored on DVD

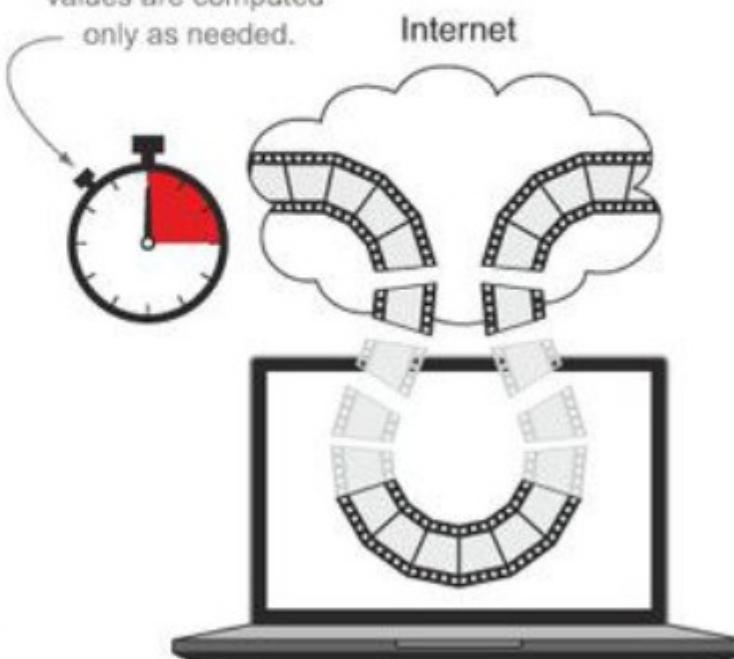
A stream in Java 8 is like a movie  
streamed over the internet.

Eager construction means  
waiting for computation  
of ALL values

Lazy construction means  
values are computed  
only as needed.



Like a DVD, a collection holds all the values that  
the data structure currently has—every element in  
the collection has to be computed before it  
can be added to the collection.



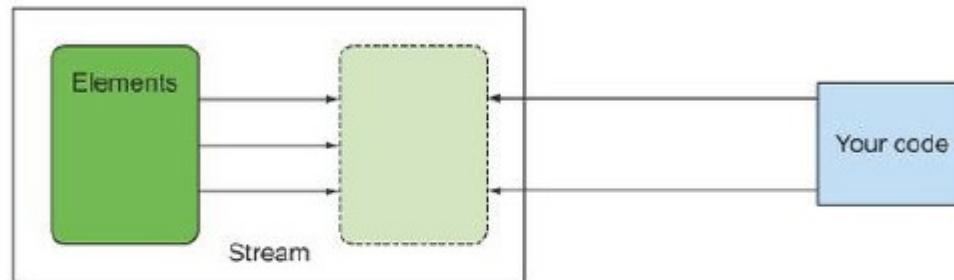
Like a streaming video, values  
are computed as they are needed.

# 3. internal vs external iteration

## Internal vs. external iteration

Stream

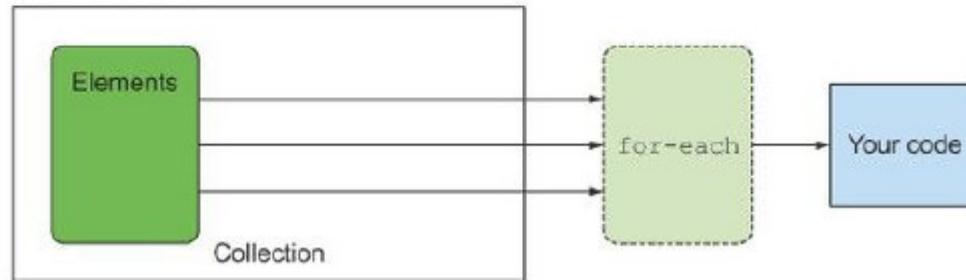
Internal iteration



---

Collection

External iteration



# 4. Intermediate operations

## Intermediate operations

| Operation | Type         | Return type | Argument of the operation | Function descriptor             |
|-----------|--------------|-------------|---------------------------|---------------------------------|
| filter    | Intermediate | Stream<T>   | Predicate<T>              | $T \rightarrow \text{boolean}$  |
| map       | Intermediate | Stream<R>   | Function<T, R>            | $T \rightarrow R$               |
| limit     | Intermediate | Stream<T>   |                           |                                 |
| sorted    | Intermediate | Stream<T>   | Comparator<T>             | $(T, T) \rightarrow \text{int}$ |
| distinct  | Intermediate | Stream<T>   |                           |                                 |

# 5. terminal operation

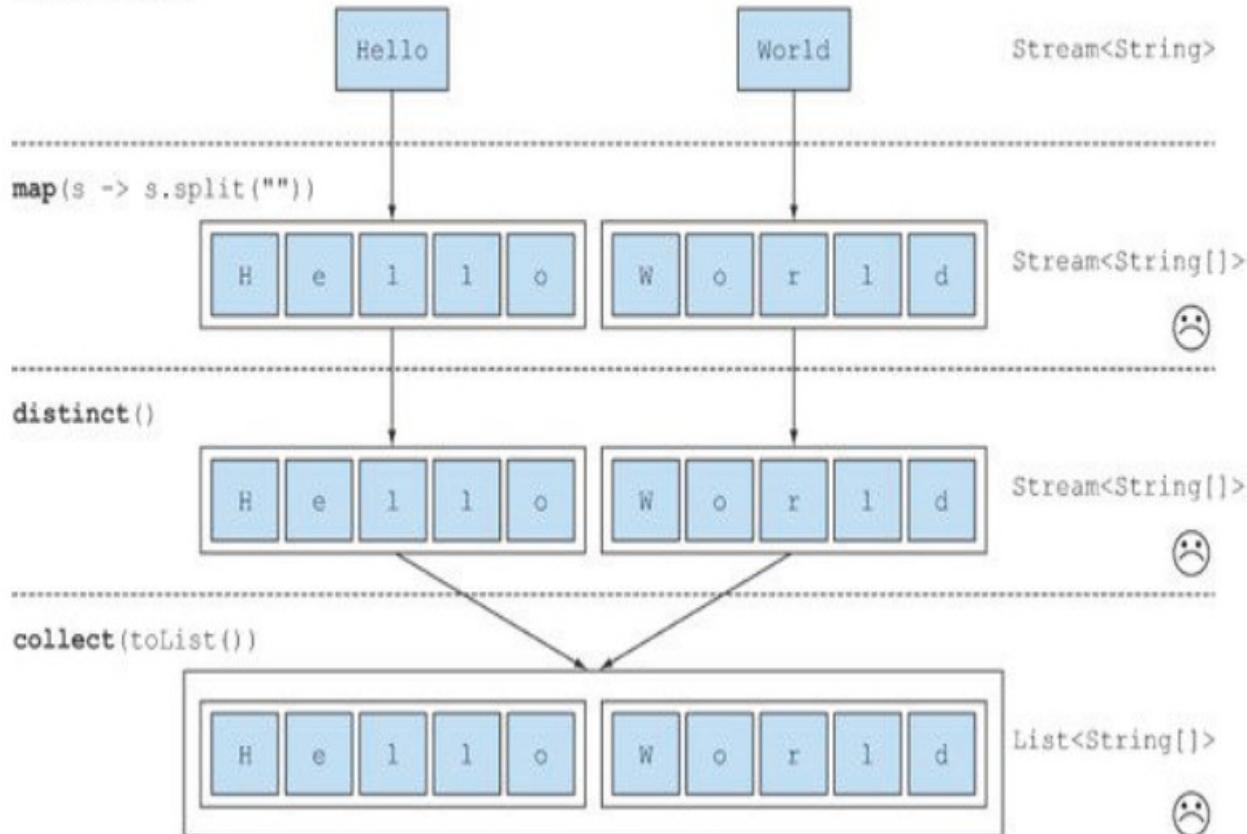
## Terminal operations

| Operation | Type     | Purpose                                                                                                  |
|-----------|----------|----------------------------------------------------------------------------------------------------------|
| forEach   | Terminal | Consumes each element from a stream and applies a lambda to each of them.<br>The operation returns void. |
| count     | Terminal | Returns the number of elements in a stream. The operation returns a long.                                |
| collect   | Terminal | Reduces the stream to create a collection such as a List, a Map, or even an Integer.                     |

# 6. need of flatMap

## Incorrect implementation : need of flatMap

Stream of words

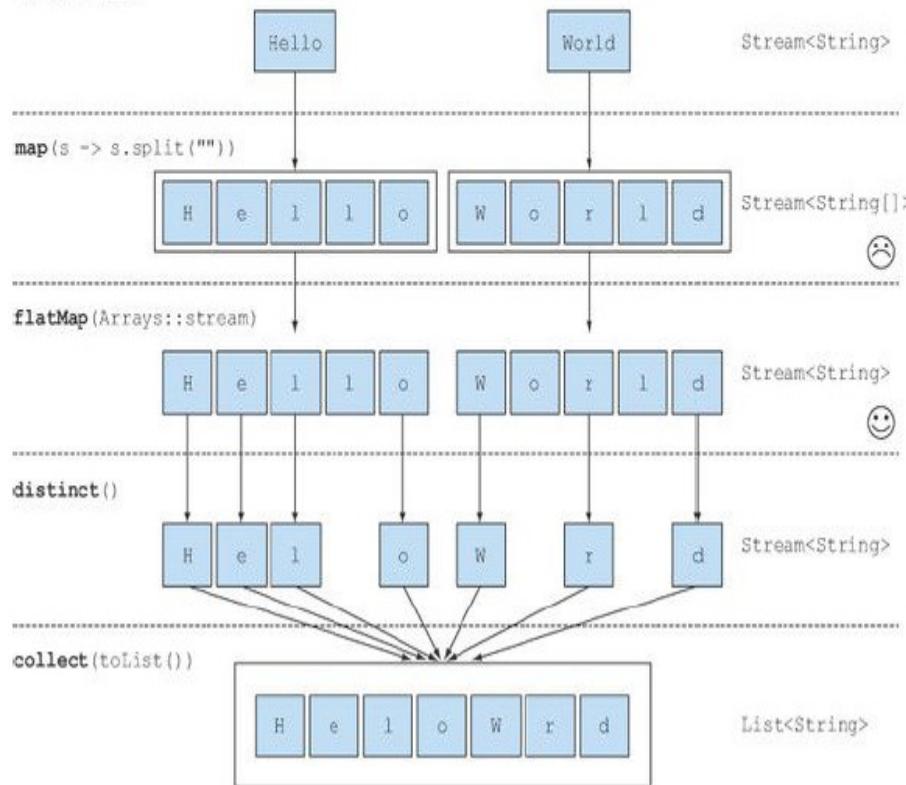


```
words.stream()
    .map(word -> word.split(""))
    .distinct()
    .collect(toList());
```

# 7. correct implementation of flatMap

```
List<String> uniqueCharacters =
words.stream()
    .map(w -> w.split(""))
    .flatMap(Arrays::stream)
    .distinct()
    .collect(Collectors.toList());
```

Stream of words



Converts each word into an array of its individual letters

Flattens each generated stream into a single stream

## Correct implementation using flatMap:

Using the flatMap method has the effect of mapping each array not with a stream but *with the contents of that stream*. All the separate streams that were generated when using map(Arrays::stream) get amalgamated—flattened into a single stream

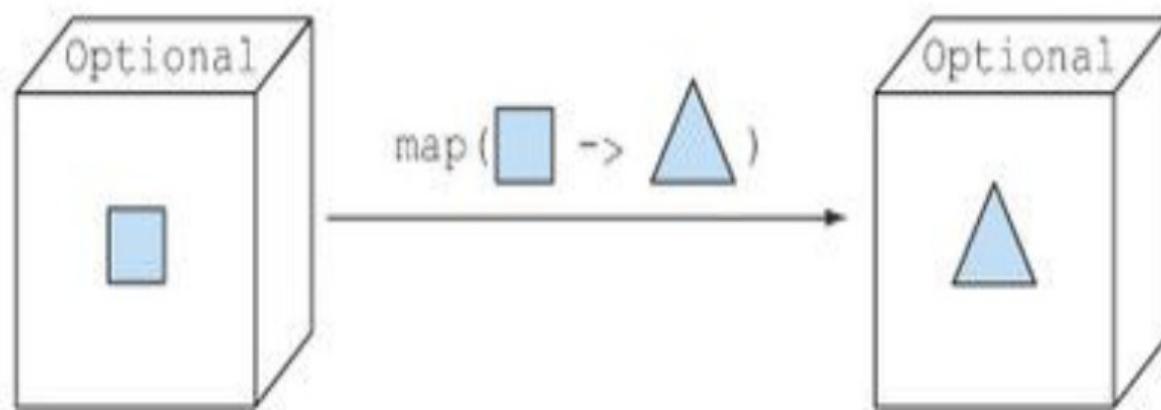
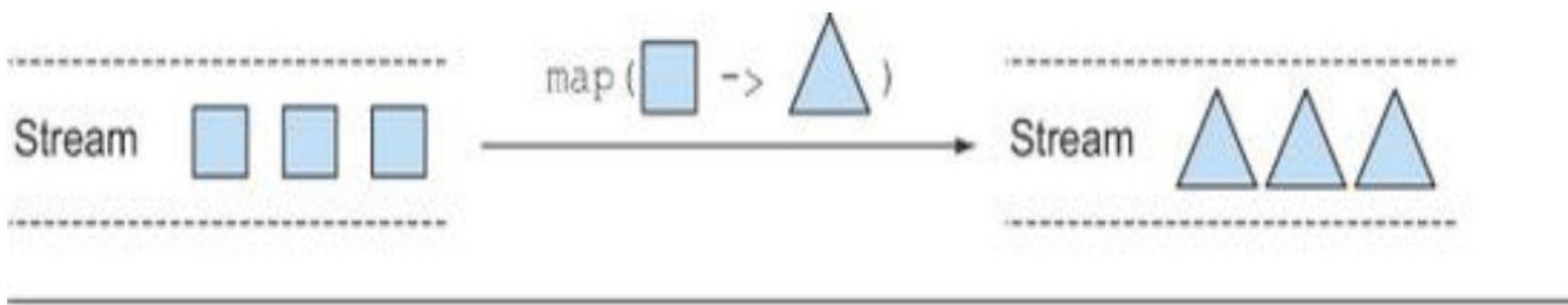
8. list of all Intermediate  
and terminal operations

## Intermediate and terminal operations

| Operation | Type                                 | Return type | Type/functional interface used | Function descriptor | Operation | Type                           | Return type | Type/functional interface used | Function descriptor |
|-----------|--------------------------------------|-------------|--------------------------------|---------------------|-----------|--------------------------------|-------------|--------------------------------|---------------------|
| filter    | Intermediate                         | Stream<T>   | Predicate<T>                   | T -> boolean        | forEach   | Terminal                       | void        | Consumer<T>                    | T -> void           |
| distinct  | Intermediate<br>(stateful-unbounded) | Stream<T>   |                                |                     | collect   | terminal                       | R           | Collector<T, A, R>             |                     |
| skip      | Intermediate<br>(stateful-bounded)   | Stream<T>   | long                           |                     | reduce    | Terminal<br>(stateful-bounded) | Optional<T> | BinaryOperator<T>              | (T, T) -> T         |
| limit     | Intermediate<br>(stateful-bounded)   | Stream<T>   | long                           |                     | count     | Terminal                       | long        |                                |                     |
| map       | Intermediate                         | Stream<R>   | Function<T, R>                 | T -> R              | findAny   | Terminal                       | Optional<T> |                                |                     |
| flatMap   | Intermediate                         | Stream<R>   | Function<T, Stream<R>>         | T -> Stream<R>      | findFirst | Terminal                       | Optional<T> |                                |                     |
| sorted    | Intermediate<br>(stateful-unbounded) | Stream<T>   | Comparator<T>                  | (T, T) -> int       |           |                                |             |                                |                     |
| anyMatch  | Terminal                             | boolean     | Predicate<T>                   | T -> boolean        |           |                                |             |                                |                     |
| noneMatch | Terminal                             | boolean     | Predicate<T>                   | T -> boolean        |           |                                |             |                                |                     |
| allMatch  | Terminal                             | boolean     | Predicate<T>                   | T -> boolean        |           |                                |             |                                |                     |

# 9. comparing map methods of streams and optional

## Comparing the map methods of Streams and Optionals



# Background: Need of Lambda expression

- ▶ Anonymous inner classes let define a functional behaviour, but with a lot of code

```
JButton testButton = new JButton("Test Button");

testButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent ae) {
        System.out.println("Click!");
    }
});
```

- ▶ All we need is a way to write only the code in the method.



# Lambda expressions

- ▶ provide a clear and concise way to represent a “one abstract method interface” (a so-called **functional interface**) using an expression

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(event -> System.out.println("Click!"));
```

- ▶ A lambda is composed of three parts:

| Argument list | Arrow token | Body of the method           |
|---------------|-------------|------------------------------|
| event         | →           | System.out.println("Click!") |

- ▶ It works because the listener has only one abstract method and the compiler can infer what to do from the interface

```
package java.awt.event;

import java.util.EventListener;

public interface ActionListener extends EventListener {
    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);
}
```

# Lambda expressions

## ► Variations in signatures

| Signature                              | Argument list | Arrow token | Body of the method          |
|----------------------------------------|---------------|-------------|-----------------------------|
| void execute()                         | ()            | →           | System.out.println("foo!")  |
| String getString()                     | ()            | →           | "foo"                       |
| Integer increment(Integer value)       | (value)       | →           | new Integer(value +1)       |
| String concatenate(String a, String b) | (a, b)        | →           | a.toString() + b.toString() |
| void process(T t)                      | (t)           | →           | {}                          |



# Data setup

---

```
public class Book {  
    private List<Author> authors;  
    private String title;  
    private int pages;  
    private Genre genre;  
    private int year;  
    private String Isbn;  
}
```

```
public class Author {  
    private String name;  
    private String lastName;  
    private String country;  
}  
  
public enum Genre {  
    NOVEL, SHORT_NOVEL, NON_FICTION;  
}
```



# Lambda sample

- We need to find the books with more than 400 pages.

```
public List getLongBooks(List books) {  
    List accumulator = new ArrayList<>();  
    for (Book book : books) {  
        if (book.getPages() > 400) {  
            accumulator.add(book);  
        }  
    }  
    return accumulator;  
}
```

- Now the requirements has changed and we also need to filter for genre of the book

```
public List getLongNonFictionBooks(List books) {  
    List accumulator = new ArrayList<>();  
    for (Book book : books) {  
        if (book.getPages() > 400 && Genre.NON_FICTION.equals(book.getGenre())) {  
            accumulator.add(book);  
        }  
    }  
    return accumulator;  
}
```

- We need a different method for every filter, while the only change is the if condition



# Lambda sample

- We can use a lambda. First we define a functional interface, which is an interface with only one abstract method

```
@FunctionalInterface  
public interface BookFilter {  
  
    public boolean test(Book book);  
}
```

- Then use this functional interface to create a book filter and write as many implementation we want in just one line

```
public static List lambdaFilter(List books, BookFilter bookFilter) {  
    List accumulator = new ArrayList<>();  
    for (Book book : books) {  
        if (bookFilter.test(book)) {  
            accumulator.add(book);  
        }  
    }  
    return accumulator;  
}  
  
// one line filters  
List longBooks = lambdaFilter(Setup.books, b -> b.getPages() > 400);  
  
BookFilter nflbFilter = b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre();  
List longNonFictionBooks = lambdaFilter(Setup.books, nflbFilter);
```



# Functional interfaces

- We don't need to write all the functional interfaces because Java 8 API defines the basic ones in *java.util.function package*

| Functional interface | Descriptor                          | Method name |
|----------------------|-------------------------------------|-------------|
| Predicate<T>         | $T \rightarrow \text{boolean}$      | test()      |
| BiPredicate<T, U>    | $(T, U) \rightarrow \text{boolean}$ | test()      |
| Consumer<T>          | $T \rightarrow \text{void}$         | accept()    |
| BiConsumer<T, U>     | $(T, U) \rightarrow \text{void}$    | accept()    |
| Supplier<T>          | $() \rightarrow T$                  | get()       |
| Function<T, R>       | $T \rightarrow R$                   | apply()     |
| BiFunction<T, U, R>  | $(T, U) \rightarrow R$              | apply()     |
| UnaryOperator<T>     | $T \rightarrow T$                   | identity()  |
| BinaryOperator<T>    | $(T, T) \rightarrow T$              | apply()     |

- So we did not need to write the BookFilter interface, because the Predicate interface has exactly the same descriptor



# Lambda sample

- So we can rewrite our code as

```
public static List lambdaFilter(List books, Predicate bookFilter) {  
    List accumulator = new ArrayList<>();  
    for (Book book : books) {  
        if (bookFilter.test(book)) {  
            accumulator.add(book);  
        }  
    }  
    return accumulator;  
}  
  
// one line filters  
List longBooks = lambdaFilter(Setup.books, b -> b.getPages() > 400);  
  
Predicate nflbFilter = b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre();  
List longNonFictionBooks = lambdaFilter(Setup.books, nflbFilter);
```



# Lambdas and existing interfaces

- Since in JDK there are a lot of interfaces with only one abstract method, we can use lambdas also for them:

```
// Runnable interface defines void run() method
Runnable r = () -> System.out.println("I'm running!");
r.run();

// Callable defines T call() method
Callable callable = () -> "This is a callable object";
String result = callable.call();

// Comparator defines the int compare(T t1, T t2) method
Comparator bookLengthComparator = (b1, b2) -> b1.getPages() - b2.getPages();
Comparator bookAgeComparator = (b1, b2) -> b1.getYear() - b2.getYear();
```



# Method reference

- Sometimes code is more readable if we refer just to the method name instead of a lambda

| Kind of method reference           | Example           |
|------------------------------------|-------------------|
| To a static method                 | Integer::parseInt |
| To an instance method of a class   | Integer::intValue |
| To an instance method of an object | n::intValue       |
| To a constructor                   | Integer::new      |

- So  

```
Function<String, Integer> lengthCalculator = (String s) -> s.length();
```
- w  

```
Function<String, Integer> lengthCalculator = String::length;
```



# Comparators

- In former versions of Java, we had to write an anonymous inner class to specify the behaviour of a Comparator

```
Collections.sort(users, new Comparator<Author>() {  
    public int compare(Author a1, Author a2) {  
        return a1.compareTo(a2.id);  
    }  
});
```

- We can use lambda for making code more readable:

```
// now sort is a oneliner!  
Collections.sort(authors, (Author a1, Author a2) -> a1.compareTo(a2));
```



# Streams

---

- ▶ The Java Collections framework relies on the concept of external iteration, as in the example below

```
for (Book book: books) {  
    book.setYear = 1900;  
}
```

- ▶ compared to internal iteration, like the example below

```
Books.forEach(b -> book.setYear(1900));
```

- ▶ The difference is not only in code readability and maintainability, is also related to performance: the runtime can optimize the internal iteration for parallelism, lazyness or reordering the data



# Streams

- Let's see again the book filter we wrote with lambdas:

```
public static List lambdaFilter(List books, Predicate bookFilter) {  
    List accumulator = new ArrayList<>();  
    for (Book book : books) {  
        if (bookFilter.test(book)) {  
            accumulator.add(book);  
        }  
    }  
    return accumulator;  
}  
  
// one line filters  
List longBooks = lambdaFilter(Setup.books, b -> b.getPages() > 400);  
  
Predicate nflbFilter = b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre();  
List longNonFictionBooks = lambdaFilter(Setup.books, nflbFilter);
```

- We can rewrite it using streams

```
// stream based filters  
List longBooks = books.stream().filter(b -> b.getPages() > 400).collect(toList());  
  
List longNonFictionBooks =  
    books.stream().filter(b -> b.getPages() > 400 && Genre.NON_FICTION == b.getGenre())  
    .collect(toList());
```

- The code is much clearer now, because we don't need the lambdaFilter() method anymore. Let's see how it works

# Streams

---

```
List longBooks = books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- ▶ What we've done is:

- ▶ calling the stream() method on the collection, for transforming it into a stream
- ▶ calling the filter() method passing a Predicate, for filtering the elements of the stream dropping any/some of them
- ▶ calling the collect() method with the static import toList() for collecting the filtered elements and put them into a List object



# Stream operations

---

| Operation                   | Operation type | Return type |
|-----------------------------|----------------|-------------|
| filter(Predicate<T>)        | intermediate   | Stream<T>   |
| map(Function <T, R>)        | intermediate   | Stream<R>   |
| flatMap(Function <T, R>)    | intermediate   | Stream<R>   |
| distinct()                  | intermediate   | Stream<T>   |
| sorted(Comparator<T>)       | intermediate   | Stream<T>   |
| peek(Consumer<T>)           | intermediate   | Stream<T>   |
| limit(int n)                | intermediate   | Stream<T>   |
| skip(int n)                 | intermediate   | Stream<T>   |
| reduce(BinaryOperator<T>)   | terminal       | Optional<T> |
| collect(Collector<T, A, R>) | terminal       | R           |
| forEach(Consumer<T>)        | terminal       | void        |
| min(Comparator<T>)          | terminal       | Optional<T> |
| max(Comparator<T>)          | terminal       | Optional<T> |
| count()                     | terminal       | long        |
| anyMatch(Predicate<T>)      | terminal       | boolean     |
| allMatch(Predicate<T>)      | terminal       | boolean     |
| noneMatch(Predicate<T>)     | terminal       | boolean     |
| findFirst()                 | terminal       | Optional<T> |
| findAny()                   | terminal       | Optional<T> |



# Optionals

- Let's start with an example: ISBN in 2007 has changed from 10 to 13 characters. To check which version of ISBN a book has we have to write

```
boolean isPre2007 = book.getIsbn().length() > 10;
```

- What if a book was published before 1970, when ISBN did not exist and the property ISBN is null? Without a proper check, NullPointerException will be thrown at runtime! Java 8 has introduced the java.util.Optional class. The code of our Book class can be now written as

```
public class Book {  
    private List<Author> authors;  
    private String title;  
    private int pages;  
    private Optional<String> Isbn;  
    private Genre genre;  
    private int year;  
}
```

# Optionals

---

- We can set the value with:

```
book.setIsbn(Optional.of("9780000000000"));
```

- Or, if the book was published before 1970:

```
book.setIsbn(Optional.empty());
```

- Or, if we don't know the value in advance:

```
book.setIsbn(Optional.ofNullable(value));
```

(in case value is null an empty Optional will be set)

- We can now get the value with

```
Optional<String> isbn = book.getIsbn();
System.out.println("Isbn: " + isbn.orElse("NOT PRESENT"));
```

- If the Optional contains an ISBN it will be returned, otherwise the string "NOT PRESENT" will be returned





### Day 3:

- **Java Stream API**

- Develop code to extract data from an object using peek() and map() methods including primitive versions of the map() method
- Search for data by using search methods of the Stream classes including findFirst, findAny, anyMatch, allMatch, noneMatch
- Develop code that uses the Optional class
- Develop code that uses Stream data methods and calculation methods
- Sort a collection using Stream API
- Save results to a collection using the collect method and group/partition data using the Collectors class
- Use flatMap() methods in the Stream API

- **Exceptions and Assertions**

- Use try-catch and throw statements
- Use catch, multi-catch, and finally clauses
- Use Autoclose resources with a try-with-resources statement
- Create custom exceptions and Auto-closeable resources
- Test invariants by using assertions

- **Use Java SE 8 Date/Time API**

- Create and manage date-based and time-based events including a combination of date and time into a single object using LocalDate, LocalTime, LocalDateTime, Instant, Period, and Duration
- Work with dates and times across timezones and manage changes resulting from daylight savings including Format date and times values
- Define and create and manage date-based and time-based events using Instant, Period, Duration, and TemporalUnit

---

# Case Study: Book Application

## Lets start...



# Data setup

---

```
public class Book {  
    private List<Author> authors;  
    private String title;  
    private int pages;  
    private Genre genre;  
    private int year;  
    private String Isbn;  
}
```

```
public class Author {  
    private String name;  
    private String lastName;  
    private String country;  
}  
  
public enum Genre {  
    NOVEL, SHORT_NOVEL, NON_FICTION;  
}
```



# Streams Problems

---

- We need all the books with more than 400 pages



# Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =  
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```



# Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =  
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- We need the top three longest books



# Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =  
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- we need the top three longest books

```
List<Book> top3LongestBooks =  
    books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).limit(3).Collect( toList());
```



# Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =  
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- We need the top three longest books

```
List<Book> top3LongestBooks =  
    books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).limit(3).Collect( toList());
```

HERE'S HOW

# Streams Problems

- We need all the books with more than 400 pages

```
List<Book> longBooks =  
    books.stream().filter(b -> b.getPages() > 400).collect(toList());
```

- We need the top three longest books

```
List<Book> top3LongestBooks =  
books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).limit(3).Collect( toList());
```

Here's how

```
List<Book> fromFourthLongestBooks =  
books.stream().sorted((b1,b2) -> b2.getPages()-b1.getPages()).skip(3).collect(toList());
```



# Streams Problems

---

- We need to get all the publishing years



# Streams Problems

---

- We need to get all the publishing years

```
List<Integer> publishingYears =  
    books.stream().map(b -> b.getYear()).distinct().collect(toList());
```



# Streams Problems

---

- We need to get all the publishing years

```
List<Integer> publishingYears =  
    books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

- We need all the authors



# Streams Problems

---

- We need to get all the publishing years

```
List<Integer> publishingYears =  
    books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

- We need all the authors

```
Set<Author> authors =  
    books.stream().flatMap(b -> b.getAuthors().stream()).distinct().collect(toSet());
```



# Streams Problems

---

- We need to get all the publishing years

```
List<Integer> publishingYears =  
    books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

- We need all the authors

```
Set<Author> authors =  
    books.stream().flatMap(b -> b.getAuthors().stream()).distinct().collect(toSet());
```



# Streams Problems

- We need to get all the publishing years

```
List<Integer> publishingYears =  
    books.stream().map(b -> b.getYear()).distinct().collect(toList());
```

- We need all the authors

```
Set<Author> authors =  
    books.stream().flatMap(b -> b.getAuthors().stream()).distinct().collect(toSet());
```

- We need all the origin countries of the authors

```
Set<String> countries =  
    books.stream().flatMap(b -> b.getAuthors().stream())  
        .map(author -> author.getCountry()).distinct().collect(toSet());
```



# Streams Problems(Hint Optional)

---

- We want the most recent published book.



# Streams Problems

---

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =  
    books.stream().min(Comparator.comparingInt(Book::getYear));
```



# Streams Problems

---

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =  
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

- We want to know if all the books are written by more than one author



# Streams Problems

---

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =  
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

- We want to know if all the books are written by more than one author

```
boolean onlyShortBooks =  
    books.stream().allMatch(b -> b.getAuthors().size() > 1);
```



# Streams Problems

---

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =  
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

- We want to know if all the books are written by more than one author

```
boolean onlyShortBooks =  
    books.stream().allMatch(b -> b.getAuthors().size() > 1);
```

- We want one of the books written by more than one author.



# Streams Problems

- We want the most recent published book.

```
Optional<Book> lastPublishedBook =  
    books.stream().min(Comparator.comparingInt(Book::getYear));
```

- We want to know if all the books are written by more than one author

```
boolean onlyShortBooks =  
    books.stream().allMatch(b -> b.getAuthors().size() > 1);
```

- We want one of the books written by more than one author.

```
Optional<Book> multiAuthorBook =  
    books.stream().filter((b -> b.getAuthors().size() > 1)).findAny();
```



# Streams Problems (Hint reduction)

---

- We want the total number of pages published.



# Streams Problems

- We want the total number of pages published.

```
Integer totalPages =  
    books.stream().map(Book::getPages).reduce(0, (b1, b2) -> b1 + b2);
```

- Or

- W 

```
Optional<Integer> totalPages =  
    books.stream().map(Book::getPages).reduce(Integer::sum);
```

 ongest book has.



# Streams Problems

- We want the total number of pages published.

```
Integer totalPages =  
    books.stream().map(Book::getPages).reduce(0, (b1, b2) -> b1 + b2);
```

- Or

```
Optional<Integer> totalPages =  
    books.stream().map(Book::getPages).reduce(Integer::sum);
```

- We want to know how many pages the longest book has.

```
Optional<Integer> longestBook =  
    books.stream().map(Book::getPages).reduce(Integer::max);
```



# The Collector interface

- The Collector interface was introduced to give developers a set of methods for reduction operations

| Method           | Return type           |
|------------------|-----------------------|
| toList()         | List<T>               |
| toSet()          | Set<T>                |
| toCollection()   | Collection<T>         |
| counting()       | Long                  |
| summingInt()     | Long                  |
| averagingInt()   | Double                |
| joining()        | String                |
| maxBy()          | Optional<T>           |
| minBy()          | Optional<T>           |
| reducing()       | ...                   |
| groupingBy()     | Map<K, List<T>>       |
| partitioningBy() | Map<Boolean, List<T>> |



# Collector Streams Problems

---

- We want the average number of pages of the books.



# Collector Streams Problems

- We want the average number of pages of the books.

```
Double averagePages =  
    books.stream().collect(averagingInt(Book::getPages));
```



# Collector Streams Problems

---

- We want the average number of pages of the books.

```
Double averagePages =  
    books.stream().collect(averagingInt(Book::getPages));
```

- We want all the titles of the books



# Collector Streams Problems

- We want the average number of pages of the books.

```
Double averagePages =  
    books.stream().collect(averagingInt(Book::getPages));
```

- We want all the titles of the books

```
String allTitles =  
    books.stream().map(Book::getTitle).collect(joining(", "));
```

- We want the second name of all the authors?



# Collector Streams Problems

- We want the average number of pages of the books.

```
Double averagePages =  
    books.stream().collect(averagingInt(Book::getPages));
```

- We want all the titles of the books

```
String allTitles =  
    books.stream().map(Book::getTitle).collect(joining(", "));
```

- We want the book with the highest number of authors?

```
Optional<Book> higherNumberOfAuthorsBook =  
    books.stream().collect(maxBy(comparing(b -> b.getAuthors().size())));
```



# Stream grouping Problems

---

- We want a Map of book per year.



# Stream grouping Problems

---

- We want a Map of book per year.

```
Map<Integer, List<Book>> booksPerYear =  
    Setup.books.stream().collect(groupingBy(Book::getYear));
```

- We want a Map of how many books are published per year per genre.



# Stream grouping Problems

- We want a Map of book per year.

```
Map<Integer, List<Book>> booksPerYear =  
    Setup.books.stream().collect(groupingBy(Book::getYear));
```

- We want a map of how many books are published per year per genre.

```
Map<Integer, Map<Genre, List<Book>>> booksPerYearPerGenre =  
    Setup.books.stream().collect(groupingBy(Book::getYear, groupingBy(Book::getGenre)));
```

- We want to count how many books are published per year.



# Stream grouping Problems

- We want a Map of book per year.

```
Map<Integer, List<Book>> booksPerYear =  
    Setup.books.stream().collect(groupingBy(Book::getYear));
```

- We want a Map of how many books are published per year per genre.

```
Map<Integer, Map<Genre, List<Book>>> booksPerYearPerGenre =  
    Setup.books.stream().collect(groupingBy(Book::getYear, groupingBy(Book::getGenre)));
```

per year.

```
Map<Integer, Long> bookCountPerYear =  
    Setup.books.stream().collect(groupingBy(Book::getYear, counting()));
```



# Stream partitioning Problems

---

- We want to classify book by hardcover.



# Stream partitioning Problems

- We want to classify book by hardcover.

```
Map<Boolean, List<Book>> hardCoverBooks =  
    books.stream().collect(partitioningBy(Book::hasHardCover));
```

- We want to further classify book by genre.



# Stream partitioning Problems

- We want to classify book by hardcover.

```
Map<Boolean, List<Book>> hardCoverBooks =  
    books.stream().collect(partitioningBy(Book::hasHardCover));
```

- We want to further classify book by genre.

```
Map<Boolean, Map<Genre, List<Book>>> hardCoverBooksByGenre =  
    books.stream().collect(partitioningBy(Book::hasHardCover, groupingBy(Book::getGenre)));
```



# Stream partitioning Problems

- We want to classify book by hardcover.

```
Map<Boolean, List<Book>> hardCoverBooks =  
    books.stream().collect(partitioningBy(Book::hasHardCover));
```

- We want to further classify book by genre.

```
Map<Boolean, Map<Genre, List<Book>>> hardCoverBooksByGenre =  
    books.stream().collect(partitioningBy(Book::hasHardCover, groupingBy(Book::getGenre)));
```

► We want to count books with/without hardcover.

```
Map<Boolean, Long> count =  
    books.stream().collect(partitioningBy(Book::hasHardCover, counting()));
```

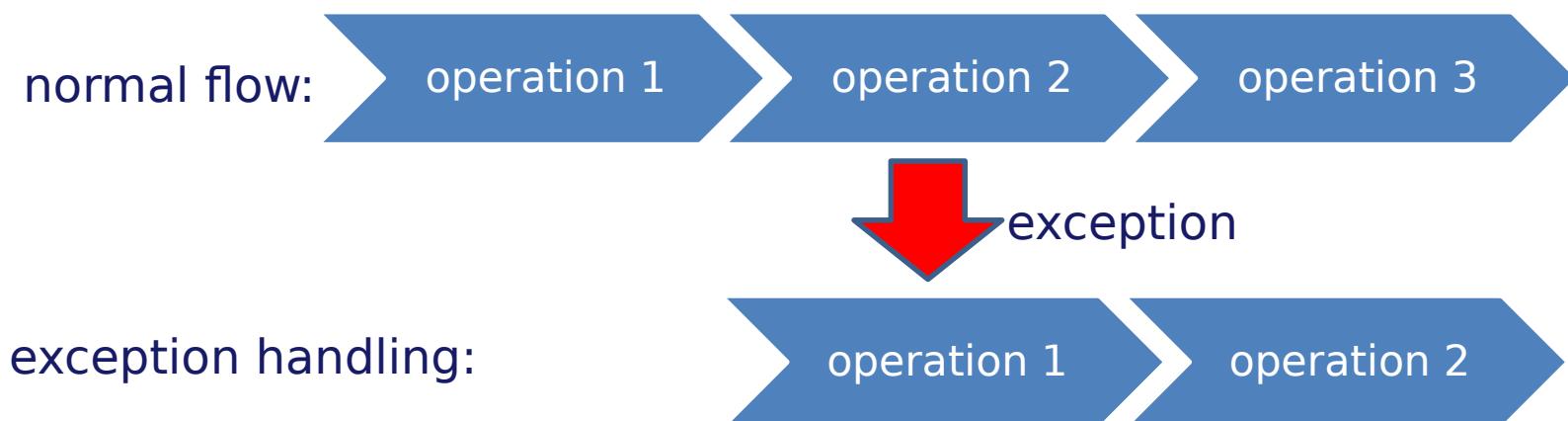




# What is Exception and Exception Handling?

**Exception** – is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

**Exception handling** is convenient way to handle errors



# Type of exceptions

## 1. Unchecked Exception

- Also called Runtime Exceptions

## 2. Checked Exception

- Also called Compile time Exceptions

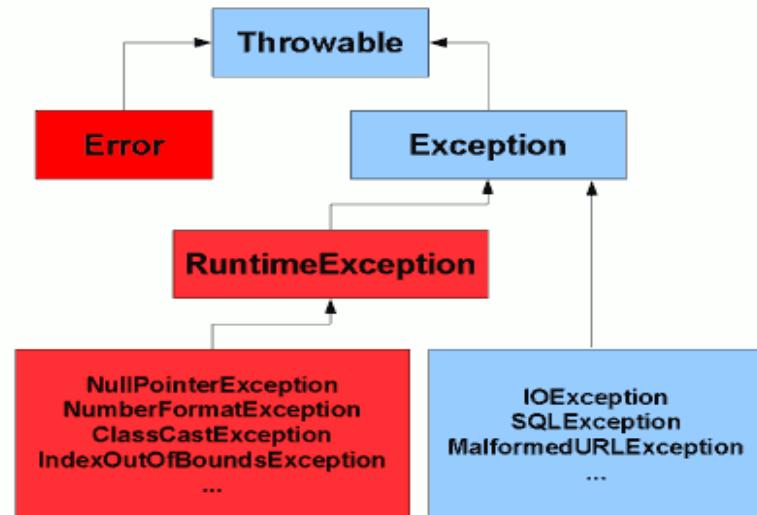
## 3. Error

- Should not be handled by programmer..like JVM crash

□ All exceptions happens at run time in programming and also in real life.....

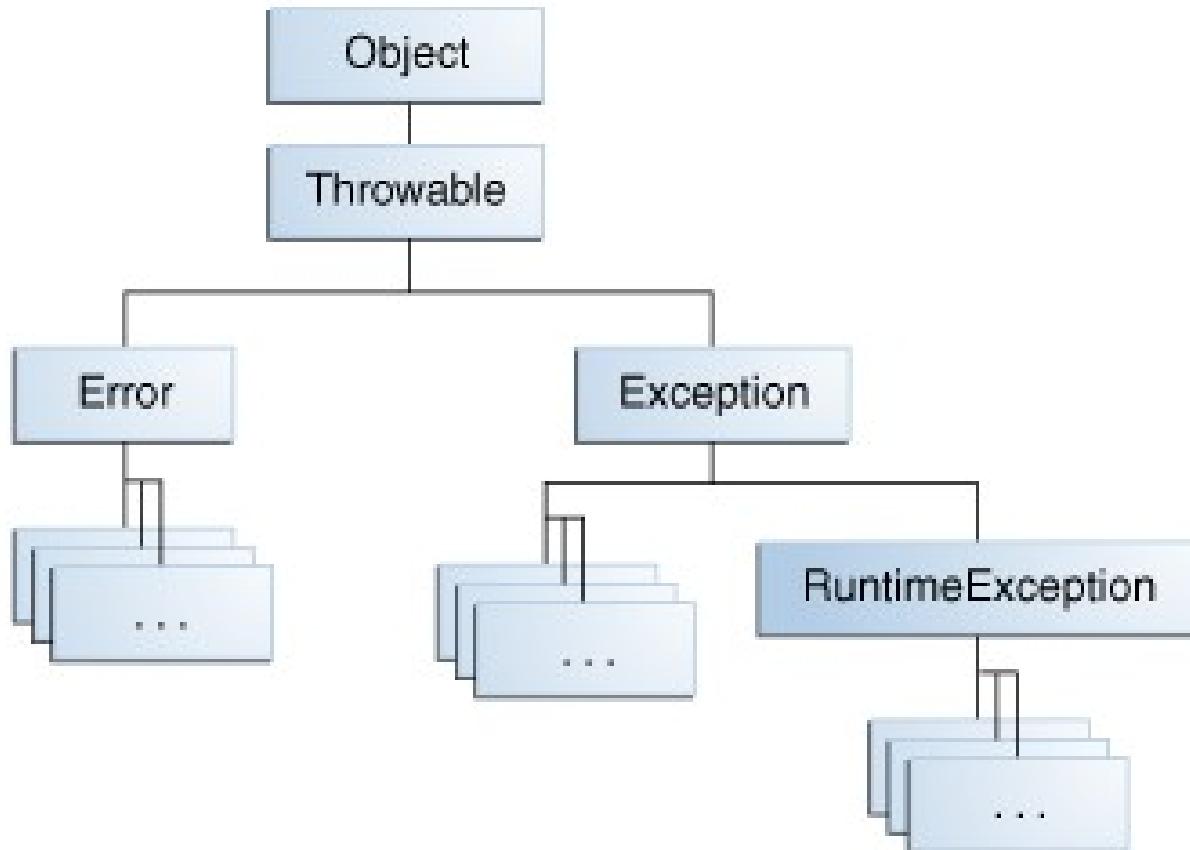
□ for checked ex, we need to tell java we know about those problems for example `readLine()` throws `IOException`

- Keywords
- Try, catch, throw, throws, finally



# Exception Class Hierarchy

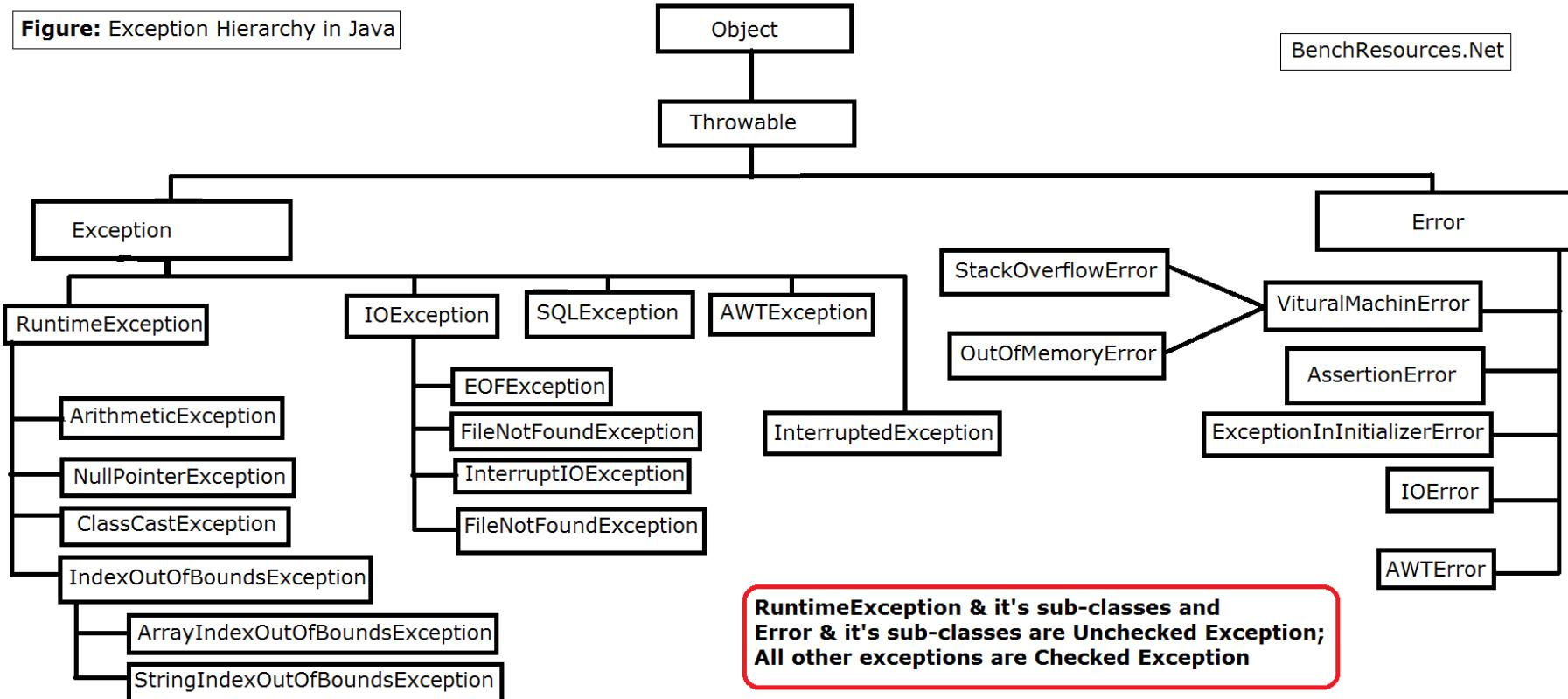
*Separate the error checking code from the main program code - the standard approach since the 1980's*



# Exception Class Hierarchy (not complete)

Figure: Exception Hierarchy in Java

BenchResources.Net



# Finally

- A finally clause is executed even if a return statement is executed in the try or catch clauses.
- An uncaught or nested exception still exits via the finally clause.
- Typical usage is to free system resources before returning from a method.

```
try {  
    // Protect one or more statements here  
}  
catch(Exception e) {  
    // Report from the exception here  
}  
finally {  
    // Perform actions here whether  
    // or not an exception is thrown  
}
```

# Java 7 Resource Management

Java 7 has introduced a new interface `java.lang.AutoCloseable` which is extended by `java.io.Closeable` interface. To use any resource in try-with-resources, it must implement `AutoCloseable` interface else java compiler will throw compilation error.

```
public class MyResource implements AutoCloseable{
    @Override
    public void close() throws Exception {
        System.out.println("Closing");
    }
}
```

```
try (MyResource sr =
      new MyResource())
{
    //doSomething with sr
}
```

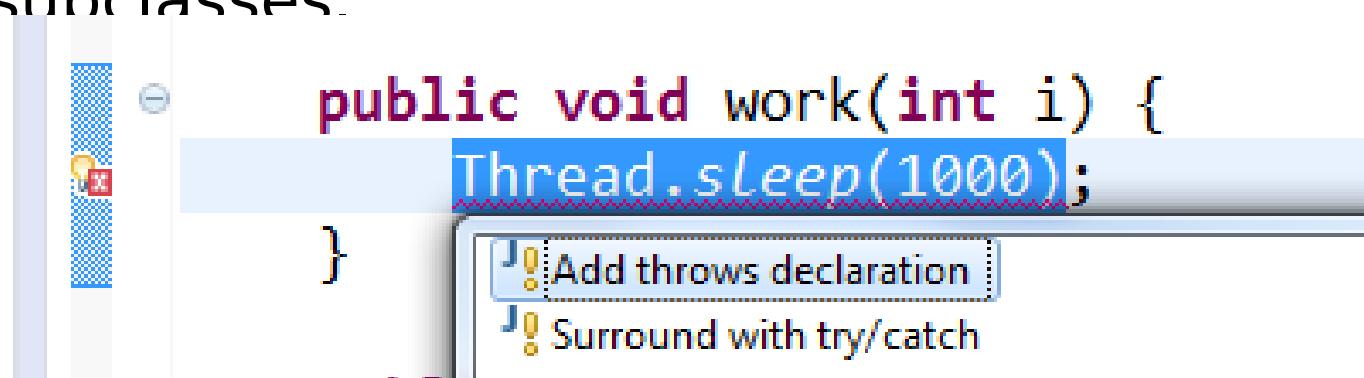
```
MyResource sr =
    new MyResource();
try {
    //doSomething with sr
} finally {
    if (sr == null) {
        sr.close();
    }
}
```

# Statement throws

If a method can throw an exception, which he does not **handle**, it must specify this behavior so that the calling code could take care of this exception.

Also there is the design **throws**, which lists the types of exceptions.

- Except Error, RuntimeException, and their subclasses.



# Statement throws

For example

```
double safeSqrt(double x)
    throws ArithmeticException {
    if (x < 0.0)
        throw new ArithmeticException();
    return Math.sqrt(x);
}
```

# Statement throw

You can throw exception using the **throw** statement

```
try {  
    MyClass myClass = new MyClass( );  
    if (myClass == null) {  
        throw new  
NullPointerException("Messages");  
    }  
} catch (NullPointerException e) {  
    // TODO  
    e.printStackTrace();  
    System.out.println(e.getMessage());  
}
```

# Summary: Dealing with Checked Exceptions

```
public static void main (String[] args) {  
    FileReader fr = new FileReader("text.txt");  
}
```

```
public static void main (String[] args) {  
    try {  
        FileReader fr = new FileReader("text.txt");  
    } catch (FileNotFoundException e) {  
        // Do something  
    }  
}
```

```
public static void main (String[] args)  
throws FileNotFoundException {  
    FileReader fr = new FileReader("text.txt");  
}
```

# Defining new exception

- You can subclass RuntimeException to create new kinds of unchecked exceptions.
- Or subclass Exception for new kinds of checked exceptions.
- Why? To improve error reporting in your program.

# Creating own checked exception

Create **checked** exception – MyException

```
// Creation subclass with two constructors
class MyException extends Exception {  
  
    // Classic constructor with a message of  
    error  
    public MyException(String msg) {  
        super(msg);  
    }  
  
    // Empty constructor  
    public MyException() { }  
}
```

# Creating own unchecked exception

If you create your own exception class from ***RuntimeException***, it's not necessary to write exception specification in the procedure.

```
class MyException extends RuntimeException { }

public class ExampleException {
    static void doSomthing(int n) {
        throw new MyException();
    }
    public static void main(String[ ] args) {
        DoSomthing(-1); // try / catch do not use
    }
}
```

# Limitation on overridden methods

- Overridden method can't change list of exceptions declared in throws section of parent method
- We can add new exception to child class when it is a descendant of an exception from the parent class or it is a runtime exception

```
public class Base {  
    public void doSomething() throws IllegalAccessException{}  
}
```

```
public class Child extends Base {  
    @Override  
    public void doSomething() throws NoSuchMethodException {}  
}
```



# Stack Trace

The exception keeps being passed out to the next enclosing block until:

- a suitable handler is found;
- or there are no blocks left to try and the program terminates with *a stack trace*

If no handler is called, then the system prints a *stack trace* as the program terminates

- it is a list of the called methods that are waiting to return when the exception occurred
- *very useful* for debugging/testing
- 

The stack trace can be printed by calling `printStackTrace()`

# Using a Stack Trace

```
// The getMessage and printStackTrace methods
public static void main( String[] args) {
    try {
        method1();
    } catch (Exception e) {
        System.err.println(e.getMessage() + "\n");
        e.printStackTrace();
    }
}
```

# Exception Handling Best Practices

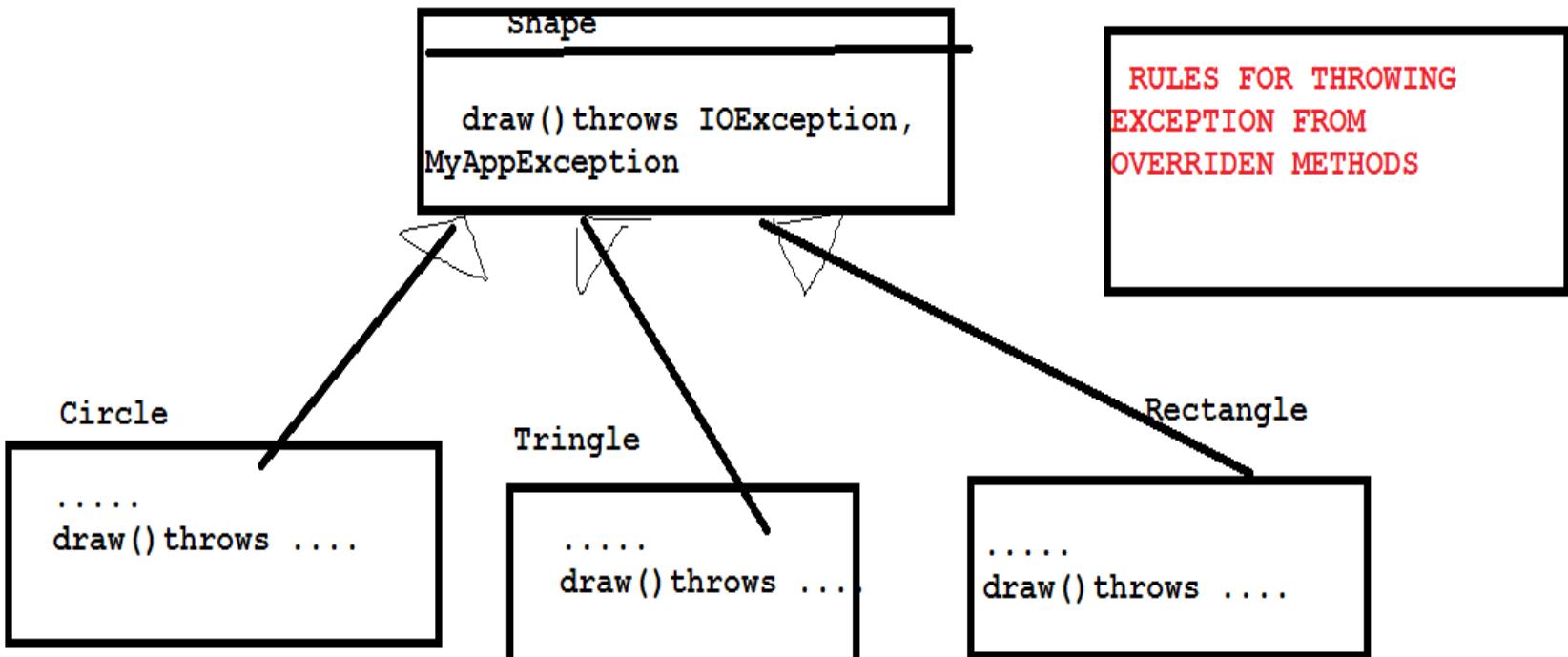
- **Use Specific Exceptions** – we should always *throw* and *catch* specific exception classes so that caller will know the root cause of exception easily and process them. This makes debugging easy and helps client application to handle exceptions appropriately.
- **Throw early** - we should try to throw exception as early as possible.
- **Catch late** – we should catch exception only when we can handle it appropriate.
- **Close resources** - we should close all the resources in finally block or use Java 7 block try-with-resources.
- **Do Not Use Exceptions to Control Application Flow**

<http://www.journaldev.com/1696/exception-handling-in-java#java-7-arm>

# Common Java Exceptions

- `ArithmaticException`
- `ArrayIndexOutOfBoundsException`
- `ArrayStoreException`
- `FileNotFoundException`
- `IOException` – general I/O failure
- `NullPointerException` – referencing a null object
- `OutOfMemoryError`
- `SecurityException` – when applet tries to perform an action not allowed by the browser's security setting.
- `StackOverflowError`
- `StringIndexOutOfBoundsException`

# Rules for throwing exception from overridden



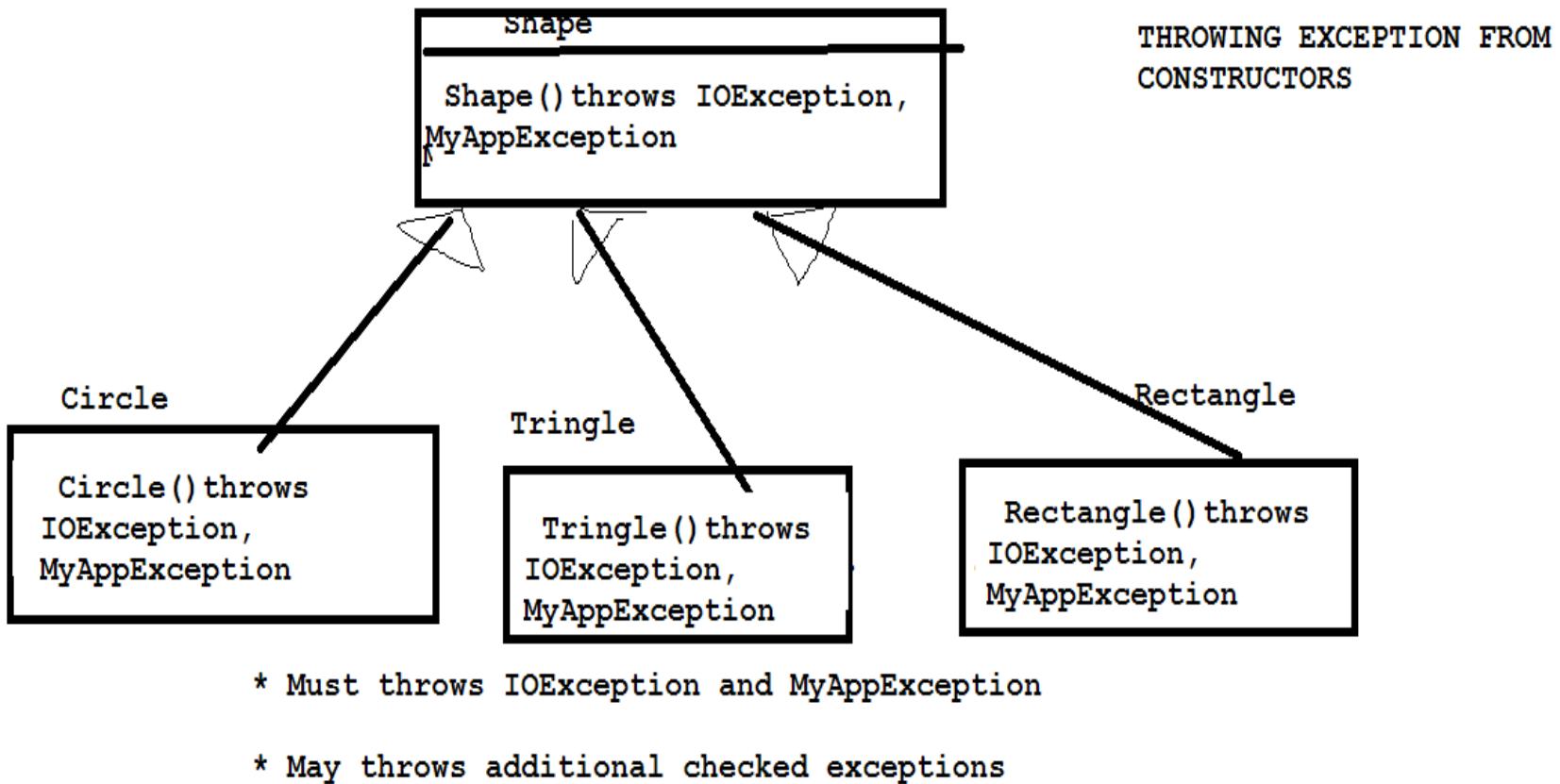
RULES FOR THROWING  
EXCEPTION FROM  
OVERRIDEN METHODS

\* Any checked exception thrown here must be subclass of  
IOException and MyAppException

\* can throw any unchecked exceptions

\*Not throwing any exception is also valid

# Rules for throwing exception from constructors



# Rules for throwing exception from overloaded and other methods

Throwing Exception from Overloaded  
and other methods...

```
class MyClass{  
    ...  
    void MyMethod(...) throws ...{}  
    void MyMethod(...,...) throws ....{}  
    void MyOtherMethod(...) throws ....{}  
}
```

\*Not an overriden method  
\*also not a const  
  
hence can throw any  
exception  
irrespective of  
exception thrown by  
other overloaded  
methods

Can throw any  
exception...

# Java 7 Resource Management

Java 7 has introduced a new interface `java.lang.AutoCloseable` which is extended by `java.io.Closeable` interface. To use any resource in try-with-resources, it must implement `AutoCloseable` interface else java compiler will throw compilation error.

```
public class MyResource implements AutoCloseable{
    @Override
    public void close() throws Exception {
        System.out.println("Closing");
    }
}
```

```
try (MyResource sr =
      new MyResource()) {
    //doSomething with sr
}
```

```
MyResource sr =
    new MyResource();
try {
    //doSomething with sr
} finally {
    if (sr == null) {
        sr.close();
    }
}
```

# • Automatic resource management

Instead of

```
public void oldTry() {  
    try {  
        fos = new  
FileOutputStream("movies.txt");  
        dos = new  
DataOutputStream(fos);  
        dos.writeUTF("Java 7 Block  
Buster");  
    } catch (IOException e) {  
        e.printStackTrace();  
    } finally {  
        try {  
            fos.close();  
            dos.close();  
        } catch (IOException e) {  
            // log the exception  
        }  
    }  
}
```

You can now say:

```
public void newTry() {  
    try (FileOutputStream fos = new  
FileOutputStream("movies.txt");  
        DataOutputStream dos = new  
DataOutputStream(fos)) {  
        dos.writeUTF("Java 7 Block  
Buster");  
    } catch (IOException e) {  
        // log the exception  
    }  
}
```

# • Multi-catch

```
public void newMultiMultiCatch() {  
    try {  
        methodThatThrowsThreeExceptions();  
    } catch (ExceptionOne e) {  
        // deal with ExceptionOne  
    } catch (ExceptionTwo | ExceptionThree e) {  
        // deal with ExceptionTwo and ExceptionThree  
    }  
}.
```



# **“Exception Handling” BEST PRACTICES**

## Exception Handling - Best Practice #1

### ❖ Handle Exceptions close to its origin

- Does NOT mean “catch and swallow” (i.e. suppress or ignore exceptions)

#### Example

```
try {  
    // code that is capable of throwing a XyzException  
} catch (XyzException e) {  
    // do nothing or simply log and proceed  
}
```

- It means, “log and handle the exception right there” or “log and throw the exception up the method call stack using a custom exception relevant to that source layer” and let it be handled later by a method up the call stack.
  - DAO layer – DataAccessException
  - Business layer – Application’s Custom Exception (example - SKUException)

## Exception Handling - Best Practice #1 (Contd..)

### Note

The approach “log and handle the exception right there” makes way to use the specific exception type to differentiate exceptions and handle exceptions in some explicit manner.

The approach “log and throw the exception up the method call stack using a custom exception relevant to that source layer” – makes way for creation of groups of exceptions and handling exceptions in a generic manner.

## Exception Handling - Best Practice #1 (Contd..)

### Note

When catching an exception and throwing it using an exception relevant to that source layer, make sure to use the construct that passes the original exception's cause. This will help preserve the original root cause of the exception.

```
try {
    // code that is capable of throwing a SQLException
} catch (SQLException e) {

    // log technical SQL Error messages, but do not pass
    // it to the client. Use user-friendly message instead

    LOGGER.error("An error occurred when searching for the SKU
details" + e.getMessage());

    throw new DataAccessException("An error occurred when searching
for the SKU details", e);
}
```

## Exception Handling - Best Practice #2

### ❖ Log Exceptions just once and log it close to its origin

Logging the same exception stack trace more than once can confuse the programmer examining the stack trace about the original source of exception. So, log Exceptions just once and log it close to its origin.

```
try {  
    // Code that is capable of throwing a XyzException  
}  
    catch (XyzException e) {  
        // Log the exception specific information.  
        // Throw exception relevant to that source layer  
    }
```

## Exception Handling - Best Practice #2 (Contd..)

### Note

There is an exception to this rule, in case of existing code that may not have logged the exception details at its origin.

In such cases, it would be required to log the exception details in the first method up the call stack that handles that exception. But care should be taken NOT to COMPLETELY overwrite the original exception's message with some other message when logging.

### Example

DAO Layer:

```
try {
    // code that is capable of throwing a SQLException
} catch (SQLException e) {
    // Note that LOGGING has been missed here
    throw new DataAccessException("An error occurred when
        processing the query.", e);
}
```

## Exception Handling - Best Practice #2 (Contd..)

Since logging was missed in the exception handler of the DAO layer, it is mandated in the exception handler of the next enclosing layer – in this example it is the (Business/Processor) layer.

Business/Processor Layer:

```
try {
    // code that is capable of throwing a DataAccessException

} catch (DataAccessException e) {
    // logging is mandated here as it was not logged
    // at its source (DAO layer method)
    LOGGER.error(e.getMessage());
    throw new SKUException(e.getMessage(), e);
}
```

## Exception Handling - Best Practice #3

### ❖ Do not catch “Exception”

#### Accidentally swallowing RuntimeException

```
try {  
    doSomething();  
} catch (Exception e) {  
    LOGGER.error(e.getMessage());  
}
```

This code

- also captures any RuntimeExceptions that might have been thrown by doSomething,
- ignores unchecked exceptions and
- prevents them from being propagated.

So, all checked exceptions should be caught and handled using appropriate catch handlers. And the exceptions should be logged and thrown to the outermost layer (i.e. the method at the top of the calling stack) using application specific custom exceptions relevant to that source layer.

## Exception Handling - Best Practice #4

### ❖ Handle Exceptions before sending response to Client

The layer of code component (i.e. the method at the top of the calling stack) that sends back response to the client, has to do the following:

- catch ALL checked exceptions and handle them by creating proper error response and send it back to client.
- NOT allow any checked exception to be “thrown” to the client.
- handle the Business layer exception and all other checked exceptions raised from within the code in that layer separately.

Examples of such components are:

- Service layer Classes in Web Service based applications
- Action Classes in Struts framework based applications

## Exception Handling - Best Practice #4 (Contd..)

### Example

```
try {  
  
    // Code that is capable of throwing a SKUException  
    // (a custom exception in this sample application)  
  
} catch (SKUException e) {  
  
    // Form error response using the exception's data,  
    // error code and/or error message  
  
}
```

## Exception Handling - Best Practice #4 (Contd..)

### An exception to handling 'Exception' – Case 1

There would be situations (although rarely) where the users would prefer a user-friendly/easy to understand message to be shown to them, instead of the system defined messages thrown by unrecoverable exceptions.

In such cases, the method at the top of the calling stack, which is part of the code that sends response to the client is expected to handle all unchecked exceptions thrown from within the 'try' block.

By doing this, technical exception messages can be replaced with generic messages that the user can understand. So, a catch handler for 'Exception' can be placed in it.

This is an exception to best practice #3 and is only for the outermost layer. In other layers downstream in the layered architecture, catching 'Exception' is not recommended for reasons explained under best practice #3.

## Exception Handling - Best Practice #4 (Contd..)

### Example

```
try {
    // Code that is capable of throwing a SKUException
    // (a custom exception in this example application)
} catch (SKUException e) {
    // Form error response using the exception's data,
    // error code and/or error message

} catch (Exception e) {

    // Log the exception related message here, since this block is
    // expected to get only the unchecked exceptions
    // that had not been captured and logged elsewhere in the code,
    // provided the exception handling and logging are properly
    // handled at the other layers in the layered architecture.

    // Form error response using the exception's data,
    // error code and/or error message
}
```

## Exception Handling - Best Practice #4 (Contd..)

### An exception to handling 'Exception' – Case 2

Certain other exceptional cases justify when it is handy and required to catch generic Exceptions. These cases are very specific but important to large, failure-tolerant systems.

Consider a request processing system that reads requests from a queue of requests and processes them in order.

```
public void processAllRequests() {  
    Request req = null;  
    try {  
        while (true) {  
            req = getNextRequest();  
            if (req != null) {  
                processRequest(req); // throws BadRequestException  
            } else { // Request queue is empty, must be done  
                break;  
            }  
        }  
    } catch (BadRequestException e) {  
        log.error("Invalid request:" + req, e);  
    }  
}
```

## Exception Handling - Best Practice #4 (Contd..)

### An exception to handling 'Exception' – Case 2 (Contd..)

With the above code, if any exception occurs while the request is being processed (either a `BadRequestException` or *any* subclass of `RuntimeException` including `NullPointerException`), then that exception will be caught *outside* the processing 'while' loop.

So, any error causes the processing loop to stop and any remaining requests *will not* be processed. That represents a poor way of handling an error during request processing.

A better way to handle request processing is to make two significant changes to the logic.

- 1) Move the try/catch block inside the request-processing loop. That way, any errors are caught and handled inside the processing loop, and they do not cause the loop to break. Thus, the loop continues to process requests, even when a single request fails.
- 2) Change the try/catch block to catch a generic `Exception`, so *any* exception is caught inside the loop and requests continue to process.

## An exception to handling 'Exception' – Case 2 (Contd..)

```
public void processAllRequests() {
    while (true) {
        Request req = null;
        try {
            req = getNextRequest();
            if (req != null) {
                processRequest(req); // throws BadRequestException
            } else { // Request queue is empty, must be done
                break;
            }
        } catch (BadRequestException e) {
            log.error("Invalid request:" + req, e);
        }
    }
}
```

## **Exception Handling - Best Practice #4 (Contd..)**

### **An exception to handling 'Exception' – Case 2 (Contd..)**

Catching a generic Exception sounds like a direct violation of the maxim suggested in best practice #3 —and it is. But the circumstance discussed is a specific and special one. In this case, the generic Exception is being caught to prevent a single exception from stopping an entire system.

In situations where requests, transactions or events are being processed in a loop, that loop needs to continue to process even when exceptions are thrown during processing.

## Exception Handling - Best Practice #5

### ❖ Handling common Runtime Exceptions

- **NullPointerException**
  - It is the developer's responsibility to ensure that no code can throw it.
  - Run CodePro and add null reference checks wherever it has been missed.
- **NumberFormatException, ParseException**

Catch these and create new exceptions specific to the layer from which it is thrown (usually from business layer) using user-friendly and non technical messages.
- To avoid **ClassCastException**, check the type of the class to be cast using the instanceof operator before casting.
- To avoid **IndexOutOfBoundsException**, check the length of the array before trying to work with an element of it.
- To avoid **ArithmaticException**, make sure that the divisor is not zero before computing the division.

## Exception Handling - Best Practice #5 (Contd..)

### ❖ Example

```
try {  
  
    int item = Integer.parseInt(itemNumber);  
  
} catch (NumberFormatException nfe) {  
  
    LOGGER.error("SKU number is invalid and not a number");  
  
    throw new SKUException("SKU number is invalid and not a number",  
        nfe);  
  
}
```

- ❖ All other unchecked exceptions (RuntimeExceptions) will be caught and handled by the 'Exception' handler in the outermost layer (as explained in Best Practice #4 - Case 1).

## Exception Handling - Best Practice #6

### ► Document Exceptions Thrown in Javadoc

For each method that throws checked exceptions, document each exception thrown with a @throws tag in its Javadoc, including the condition under which the exception is thrown.

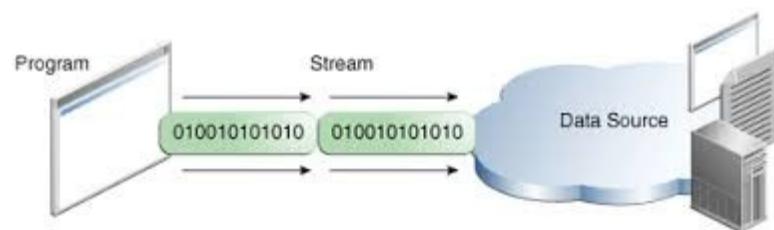


A large, stylized number 4 is centered on the page. It is composed of multiple horizontal bars of varying colors, stacked vertically. The colors include red, orange, teal, pink, yellow, and magenta. The number has a rough, textured appearance, resembling a hand-drawn or stamped digit.

4

## Day 4:

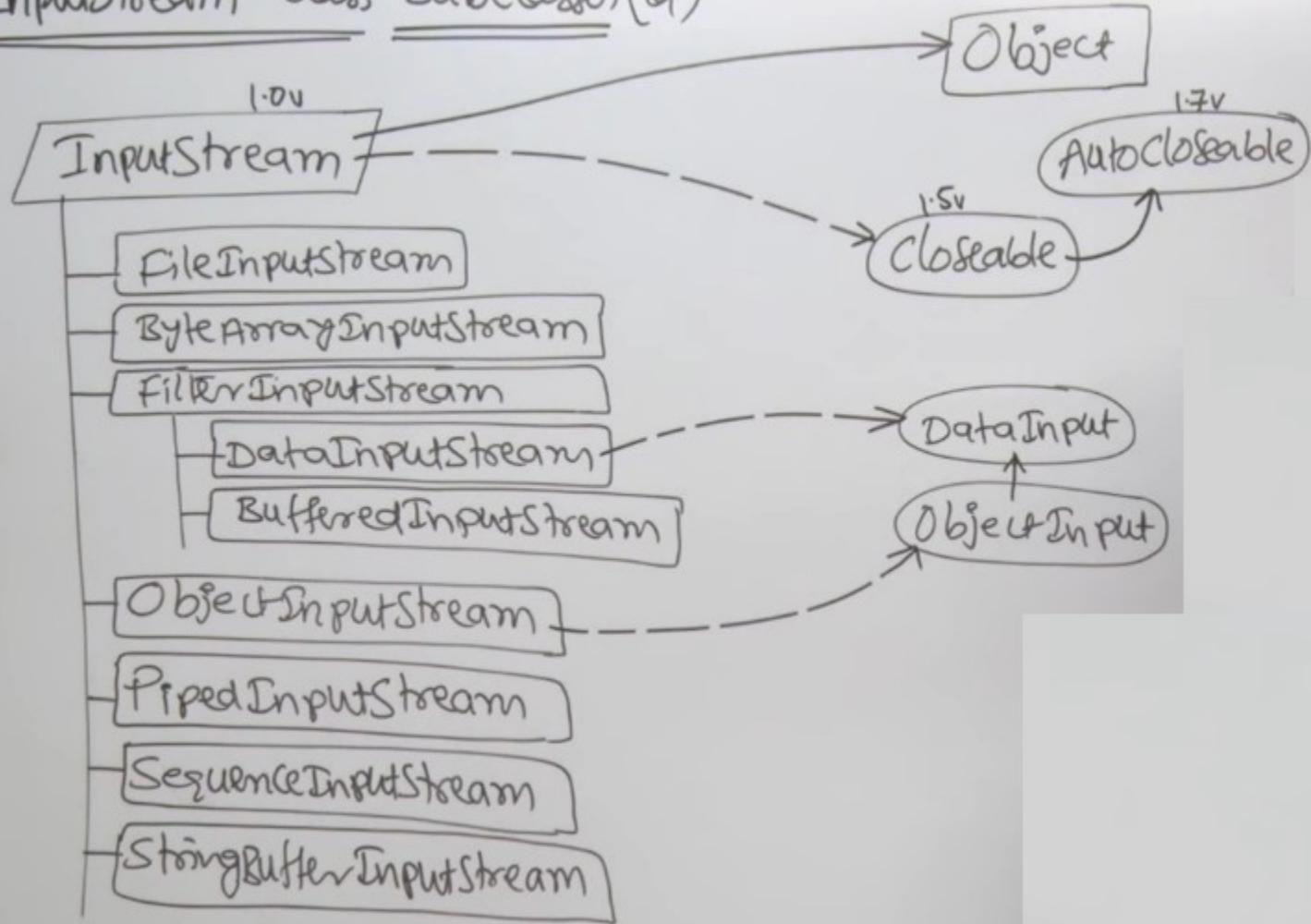
- **Java I/O Fundamentals**
  - Read and write data from the console
  - Use BufferedReader, BufferedWriter, File, FileReader, FileWriter, FileInputStream, FileOutputStream, ObjectOutputStream, ObjectInputStream, and PrintWriter in the java.io package.
- **Java File I/O (NIO.2)**
  - Use Path interface to operate on file and directory paths
  - Use Files class to check, read, delete, copy, move, manage metadata of a file or directory
  - Use Stream API with NIO.2
- **Java Concurrency**
  - Create worker threads using Runnable, Callable and use an ExecutorService to concurrently execute tasks
  - Identify potential threading problems among deadlock, starvation, livelock, and race conditions
  - Use synchronized keyword and java.util.concurrent.atomic package to control the order of thread execution
  - Use java.util.concurrent collections and classes including CyclicBarrier and CopyOnWriteArrayListUse parallel Fork/Join Framework
  - Use parallel Streams including reduction, decomposition, merging processes, pipelines and performance.



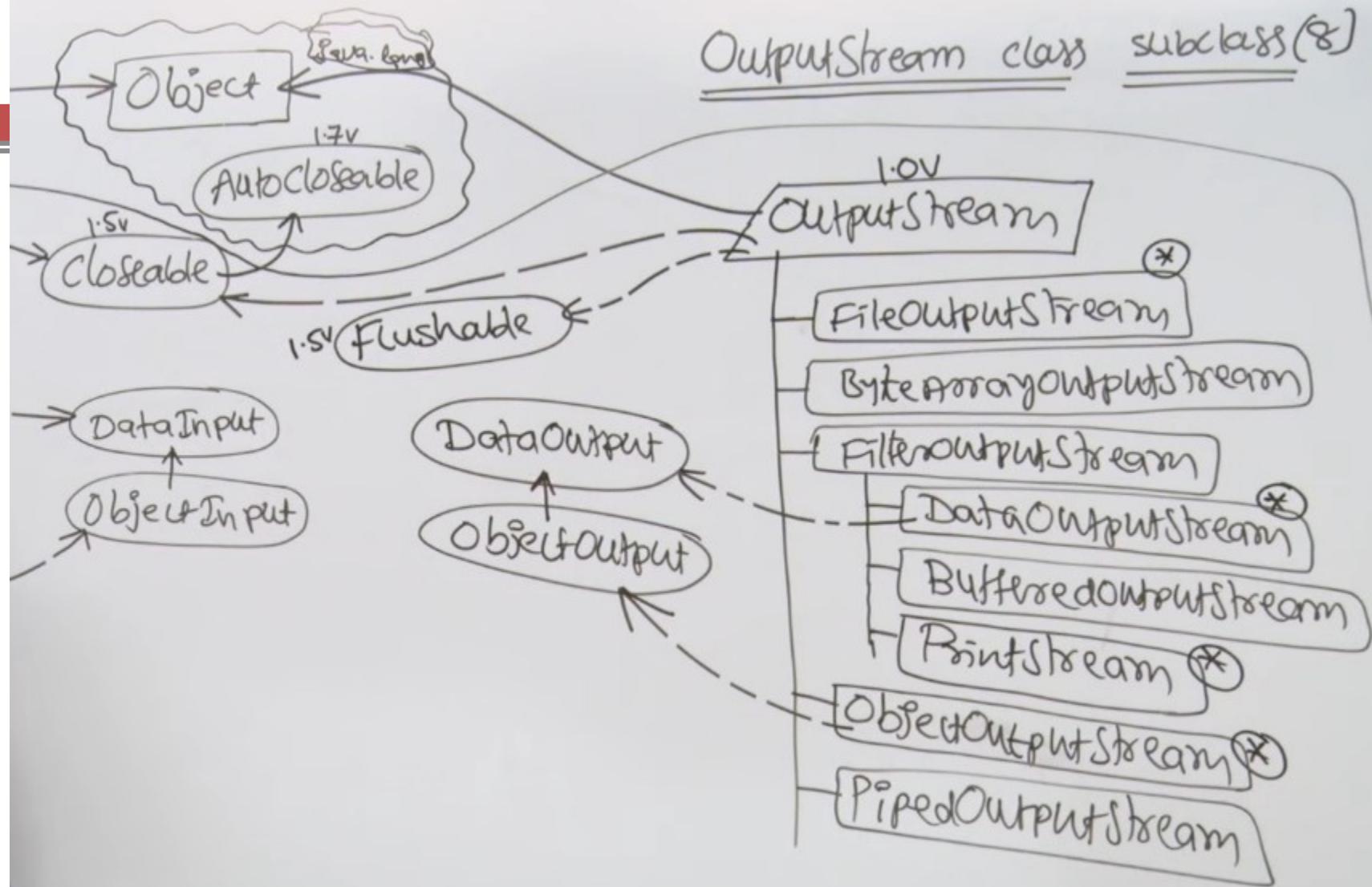
# Stream

- Stream is an abstraction that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ.
- 2 types of streams:
  - byte : for binary data
    - All byte stream class are like XXXXXXXXStream
  - character: for text data
    - All char stream class are like XXXXXXXXReader/ XXXXXXWriter

## InputStream class Subclasses(a)



## OutputStream class subclass(8)



# System class in java

- System class defined in `java.lang` package
- It encapsulate many aspect of JRE
- System class also contain 3 predefine stream variables
  - `in`
    - `System.in` (`InputStream`)
  - `out`
    - `System.out` (`PrintStream`)
  - `err`
    - `System.err` (`console`)

# File

## □ File abstraction that represent file and directories

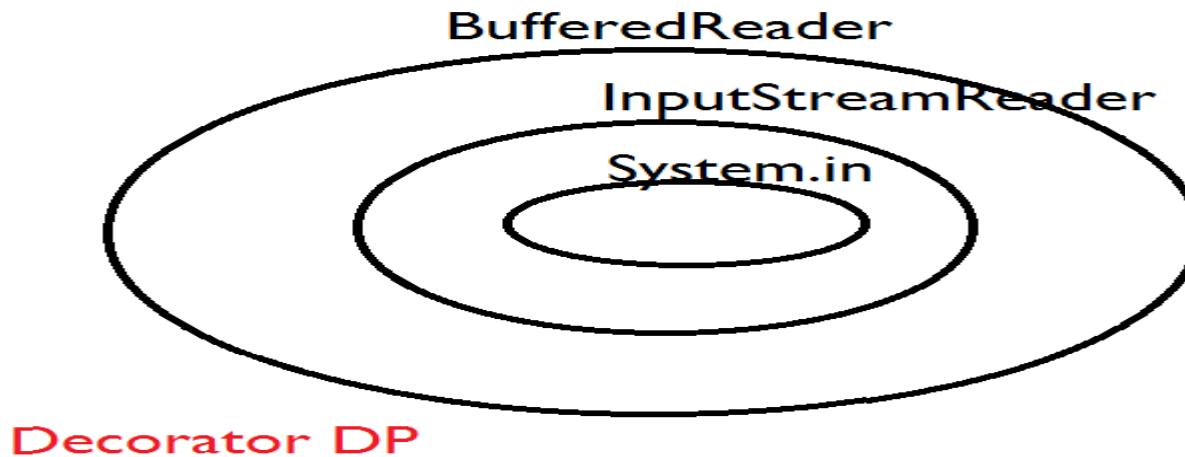
- File f=new File("....");
- boolean flag= file.createNewFile();
- boolean flag= file.mkdir();
- boolean flag=file.exists();

## □ FileWriter

```
File file = new File( "fileWrite2.txt");
FileWriter fw =new FileWriter(file);
fw.write("howdy\nfolks\n"); // write characters to
fw.flush(); // flush before closing
fw.close(); // close file when done
```

# BufferedReader and BufferedWriter

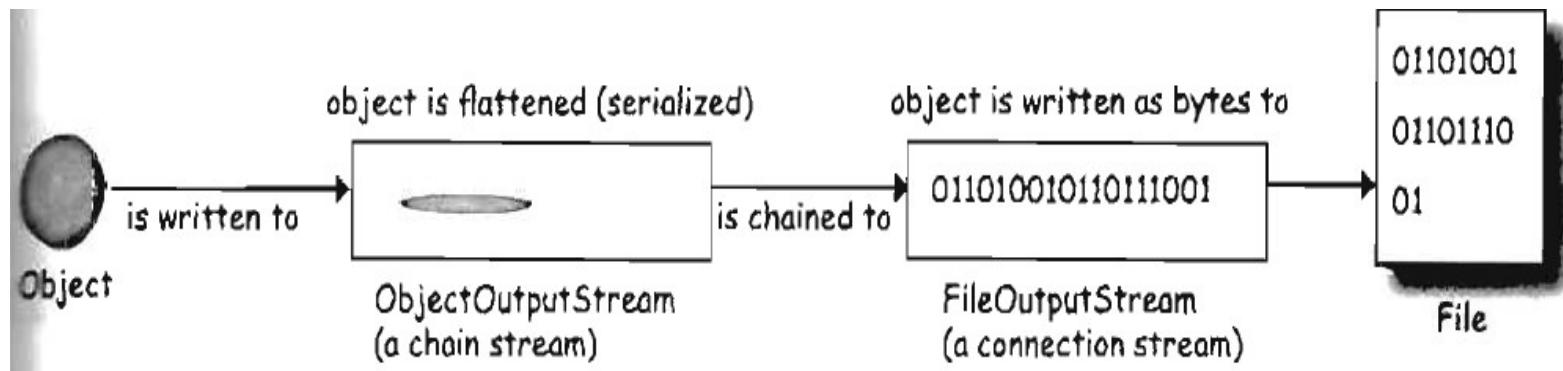
- Reading from console
  - `BufferedReader br=new BufferedReader(new InputStreamReader(System.in));`
- Reading from file
  - `BufferedReader br=new BufferedReader(new FileReader(new File("c:\\raj\\foo.txt"));`



# Serialization

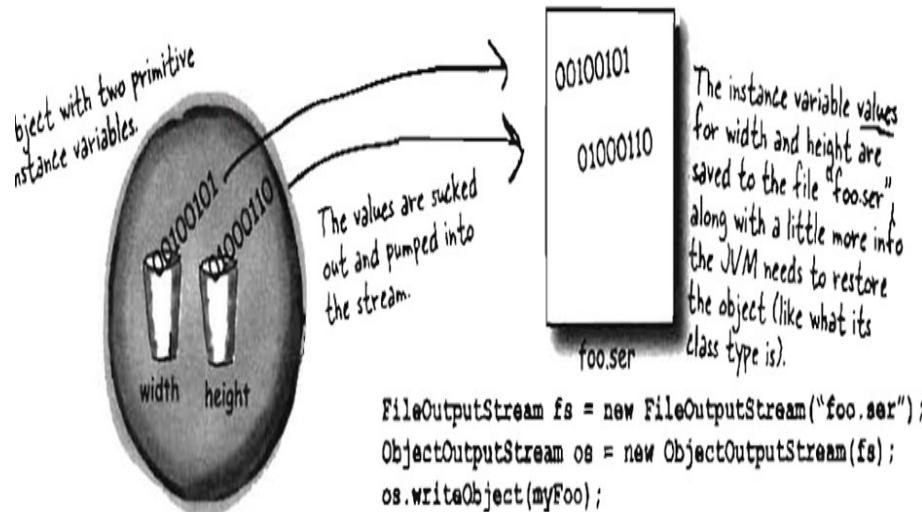


- Storing the state of the object on a file along some metadata....so that it Can be recovered back.....
- Serialization used in RMI (Remote method invocation ) while sending an object from one place to another in network...



# What actually happens during Serialization

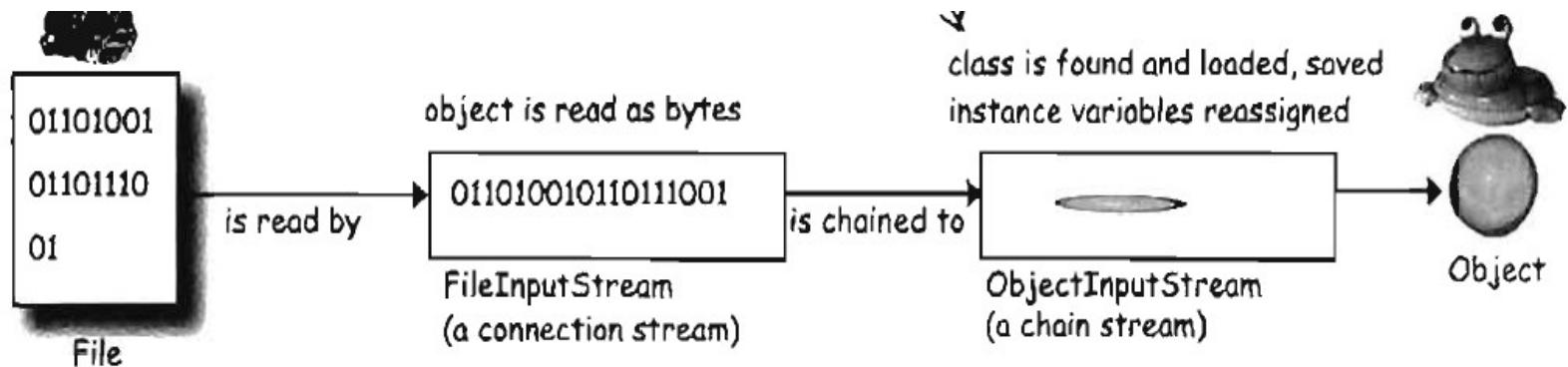
- The state of the object from heap is sucked and written along with some meta data in an file....



**When an object is serialized, all the objects it refers to from instance variables are also serialized. And all the objects those objects refer to are serialized. And all the objects those objects refer to are serialized... and the best part is, it happens automatically!**

# De- Serialization

- When an object is de-serialized, the JVM attempts to bring object back to the life by making a new object on the heap that have the same state as original object
- Transient variable don't get saved during serialization hence come with null !!!



# Hello world Example...

```
class Box implements Serializable{
    private static final long serialVersionUID = 1L;
    int l,b;

    public Box(int l, int b) {
        super();
        this.l = l;
        this.b = b;
    }

    public String toString() {
        return "Box [l=" + l + ", b=" + b + "]";
    }
}
```

```
Box box=new Box(22, 33);
```

```
FileOutputStream fo=new FileOutputStream("c:\\raj\\foofoo.ser");
ObjectOutputStream os=new ObjectOutputStream(fo);

os.writeObject(box);
```

```
box=null;//nullify to prove that object come by de-ser...
```

```
FileInputStream fi=new FileInputStream("c:\\raj\\foofoo.ser");
ObjectInputStream oi=new ObjectInputStream(fi);

box=(Box)oi.readObject();
```

```
System.out.println(box);
```

## Differences between Serialization and Externalization?

| Serialization                                                                                                                 | Externalization                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>It is Meant for Default Serialization.</b>                                                                                 | <b>It is Meant for Customized Serialization.</b>                                                                                                                                 |
| <b>Here Everything Takes Care by JVM and Programmer doesn't have any Control.</b>                                             | <b>Here Everything Takes Care by Programmer and JVM doesn't have any Control.</b>                                                                                                |
| <b>In Serialization Total Object will be Saved to the File Always whether it is required OR Not.</b>                          | <b>In Externalization Based on Our Requirement we can Save Either Total Object OR Part of the Object.</b>                                                                        |
| <b>Relatively Performance is Low.</b><br><b>Serialization is the best choice if we want to save total object to the file.</b> | <b>Relatively Performance is High.</b><br><b>Externalization is the best choice if we want to save part of the Object to the file.</b>                                           |
| <b>Serializable Interface doesn't contain any Method. It is a <i>Marker</i> Interface.</b>                                    | <b>Externalizable Interface contains 2 Methods, <i>wrtExternal()</i> and <i>readExternal()</i>. So it is Not Marker Interface.</b>                                               |
| <b>Serializable implemented Class Not required to contain public No - Argument Constructor.</b>                               | <b>Externalizable implemented Class should Compulsory contain public No - Argument Constructor. Otherwise we will get Runtime Exception Saying <i>InvalidClassException</i>.</b> |
| <b>transient Key Word will Play Role in Serialization.</b>                                                                    | <b>transient Key Word won't Play any Role in Externalization. Of Course it is Not Required.</b>                                                                                  |

# NIO-2

Why nio-2?

What is missing from java6

1. method such as delete file do not throw exception when fail
2. Rename work inconsistency
3. No Additional support of metadata
4. Inefficient file meta data access
5. file methods do not scale
6. walking tree

# NIO-2

Why nio-2?

What is missing from java6

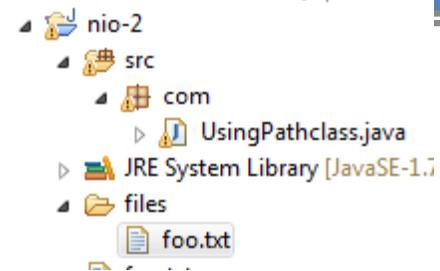
1. method such as delete file do not throw exception when fail
2. Rename work inconsistency
3. No Additional support of metadata
4. Inefficient file meta data access
5. file methods do not scale
6. walking tree

# Examples Java NIO-2

1. Using the Path class
2. Managing files and directories
3. Reading and writing text files
4. Reading and writing binary files
4. Walking the directory tree
5. Finding files.
6. Watching a directory for file changes

# Using the Path class

```
public class UsingPathclass {  
  
    public static void main(String[] args) {  
        Path path=Paths.get("files/foo.txt");  
        System.out.println(path.toString());  
        System.out.println(path.getNameCount());  
        System.out.println(path.getFileName());  
        System.out.println(path.getName(0));  
        System.out.println(path.getName(path.getNameCount()-1));  
    }  
}
```



```
<terminated> UsingPathclass [Java Application]  
files\foo.txt  
2  
foo.txt  
files  
foo.txt
```

# Managing files and directories

## CRUD

```
//copy files delete file if already exist.....  
  
Path source=Paths.get("files/foo.txt");  
Path target=Paths.get("files/temp.txt");  
Files.copy(source, target,StandardCopyOption.REPLACE_EXISTING);  
  
//Delete.....  
Path todelete=Paths.get("files/temp.txt");  
Files.delete(todelete);  
System.out.println("file deleted.....");  
  
//creating directory  
  
Path dir=Paths.get("files/newDir");  
Files.createDirectories(dir);  
  
//moving an file to dir  
//Files.move(source, target, options)|  
Files.move(source, dir.resolve(source.getFileName()), StandardCopyOption.REPLACE_EXISTING);  
\\
```

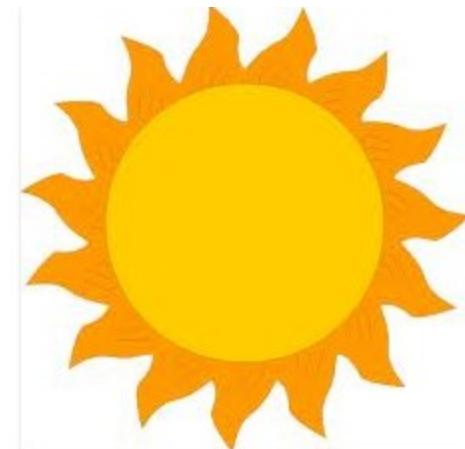
# Read/Write text files...

```
public static void main(String[] args) {
    Path source=Paths.get("files/old.txt");
    Path target=Paths.get("files/new.txt");

    Charset charset=Charset.forName("US-ASCII");
    try
    {
        BufferedReader br=Files.newBufferedReader(source,charset);
        BufferedWriter wr=Files.newBufferedWriter(target, charset);
    }
    {
        String line=null;
        while((line=br.readLine())!=null)//reading.....
        {
            System.out.println(line);
            wr.append(line,0,line.length());//writing
            wr.newLine();//will append new line
        }
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
}
```

# Reading and writing binary files

```
try
{
    FileInputStream in=new FileInputStream("files/sun.jpg");
    FileOutputStream fs=new FileOutputStream("files/new.jpg");
    //will also create an new file
}
{
    int c;
    while((c=in.read())!=-1)
    {
        fs.write(c);
    }
}
catch(IOException e)
{
    e.printStackTrace();
}
```



# Walking the directory tree

Step 1: Create own custom class extending SimpleFileVisitor class    SimpleFileVisiter class consist of  
4 callback methods

```
class MyFileVisiter extends SimpleFileVisitor<Path>
{
    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
        throws IOException {
        return super.preVisitDirectory(dir, attrs);
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException {
        return super.visitFile(file, attrs);
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc)
        throws IOException {
        return super.visitFileFailed(file, exc);
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc)
        throws IOException {
        return super.postVisitDirectory(dir, exc);
    }
}
```

```

class MyFileVisiter extends SimpleFileVisitor<Path>
{
    @Override
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
        throws IOException {
        System.out.println("About to visit: "+dir);
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException {
        if(attrs.isRegularFile())
        {
            System.out.println("Regular file:"+file);
        }
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFileFailed(Path file, IOException exc)
        throws IOException {
        System.out.println(exc.getMessage());
        return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult postVisitDirectory(Path dir, IOException exc)
        throws IOException {
        System.out.println("visited: "+dir);
        return FileVisitResult.CONTINUE;
    }
}

public class WalkingTheDirectoryTree {

    public static void main(String[] args) throws IOException {
        Path fileDir=Paths.get("files2");
        MyFileVisiter visitor=new MyFileVisiter();
        Files.walkFileTree(fileDir, visitor);
    }
}

```

## Step 2: Use MyFileVisitor in Files.walkFileTree(... .)

# Finding an particular file/dir

- Step 1
  - Create FileFinder class that extends SimpleFileVisitor<Path>
- Step 2
  - Create PathMatcher

```
class FileFinder extends SimpleFileVisitor<Path>{
    PathMatcher matcher;
    public ArrayList<Path> foundPath=new ArrayList<>();

    FileFinder(String pattern){
        matcher=FileSystems.getDefault().getPathMatcher("glob:"+pattern);
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
        throws IOException {
        Path name=file.getFileName();
        System.out.println("Examining :"+name);
        if(matcher.matches(name))
        {
            foundPath.add(file);
        }
        return FileVisitResult.CONTINUE;
    }
}
```

# Finding an particular file/dir: now testing it.....

```
Path fileDir=Paths.get("files2");
//looks for specific file
FileFinder finder= new FileFinder("file1.txt");

Files.walkFileTree(fileDir, finder);

ArrayList<Path>foundFilles=finder.foundPath;

if(foundFilles.size()>0)
{
    for(Path path:foundFilles)
    {
        System.out.println(path.toRealPath(LinkOption.NOFOLLOW_LINKS));
    }
}
```

# Watching a directory for file changes

- Monitoring Update/delete modify any file/directory in file system....
- Watch service
  - Service that monitor an directory and report when any content in that directory is deleted/modified etc

```
try{
    WatchService service=FileSystems.getDefault().newWatchService();
}
{
Map<WatchKey,Path>keyMap=new HashMap<>();
Path path=Paths.get("files2");

keyMap.put(path.register(service, StandardWatchEventKinds.ENTRY_CREATE,
    StandardWatchEventKinds.ENTRY_DELETE,StandardWatchEventKinds.ENTRY_MODIFY), path);

WatchKey watchKey;

do{

    watchKey=service.take();

    Path eventDir=keyMap.get(watchKey);
    for(WatchEvent<?>event:watchKey.pollEvents())
    {
        WatchEvent.Kind<?>kind=event.kind();

        Path eventPath=(Path)event.context();
        System.out.println(eventDir + " :" + kind+ " :" +eventPath);
    }

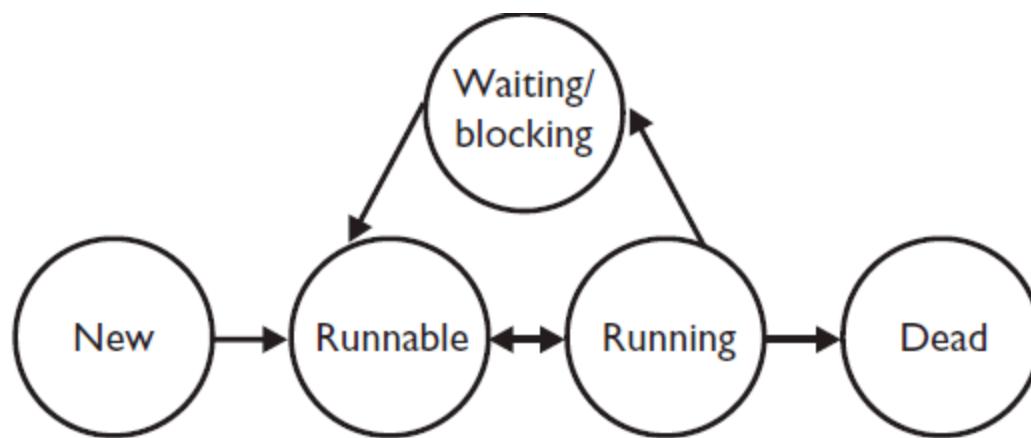
}while(watchKey.reset());

}

catch(Exception ex){

}

}
```



**How you designed it**

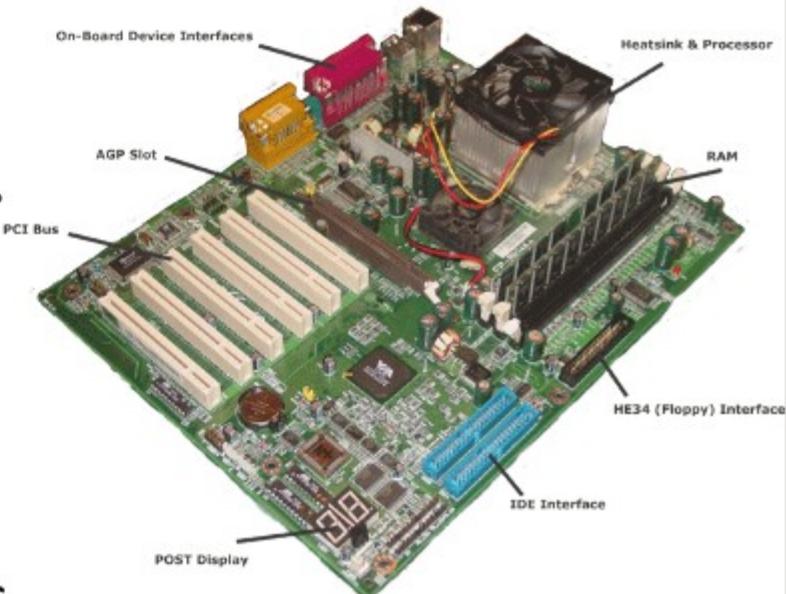


**What happens in reality**

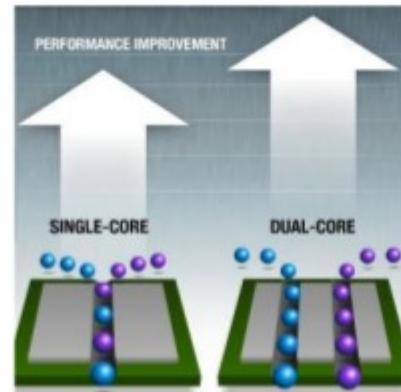
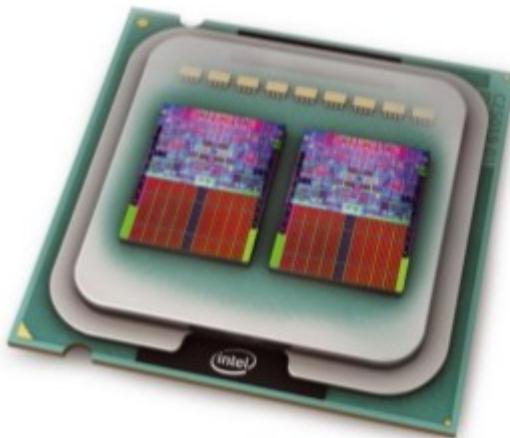


# How Concurrency Achieved?

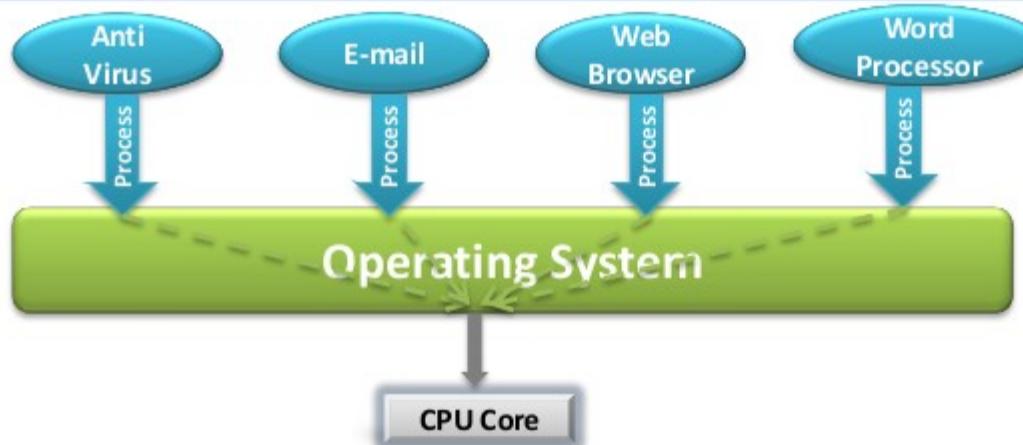
- Single Processor:
  - Has only one actual processor.
  - Concurrent tasks are often multiplexed or multi tasked.
- Multi Processor.
  - Has more than one processors.
  - Concurrent tasks are executed on different processors.



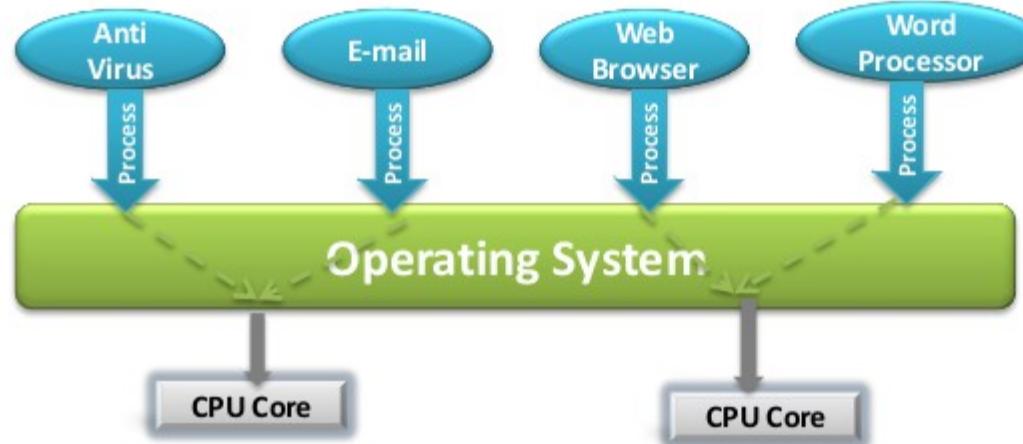
- Both types of systems can have more than one core per processor (Multicore).
- Multicore processors behave the same way as if you had multiple processors



# Cores

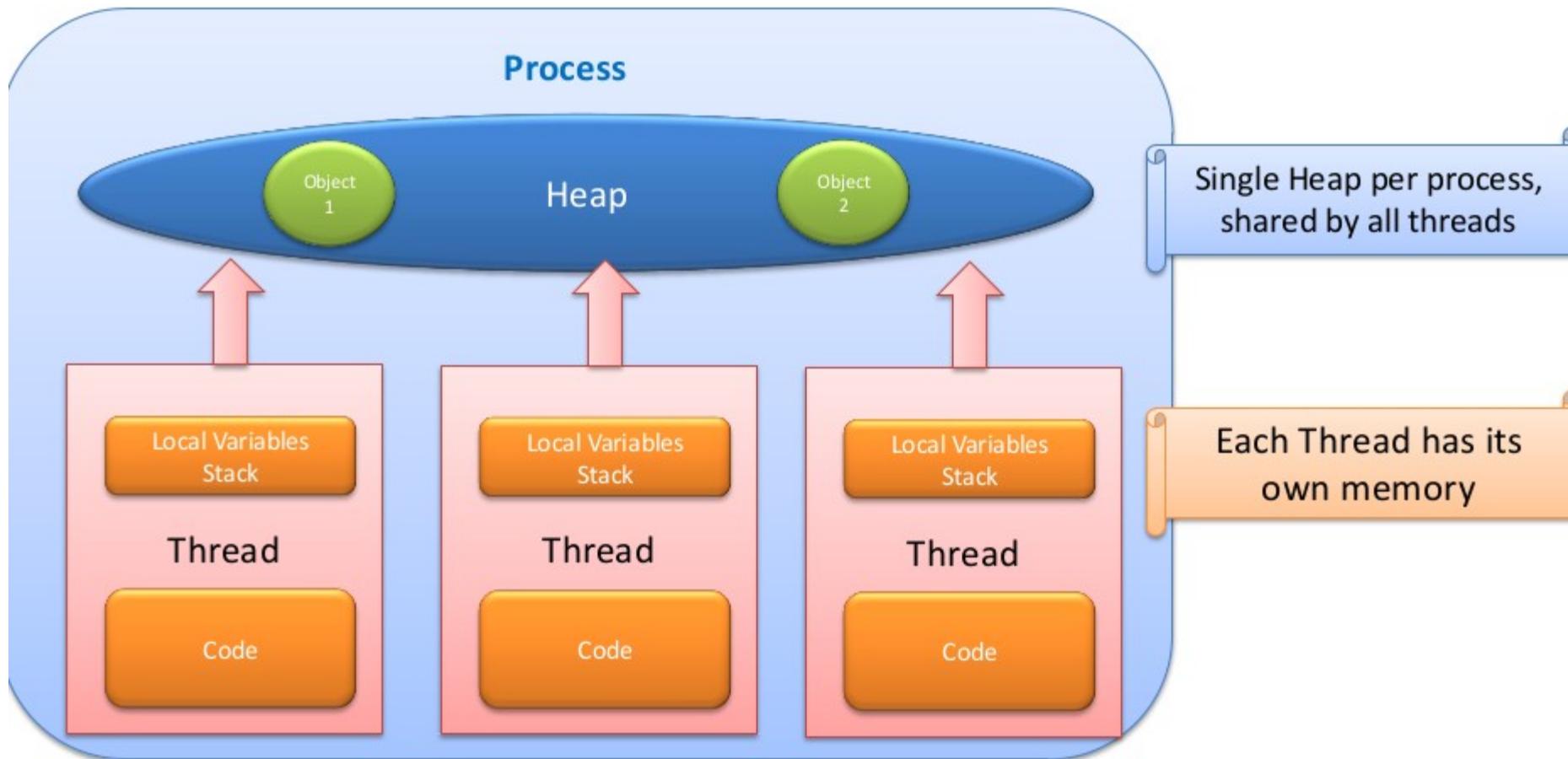


Single-core systems schedule tasks on 1 CPU to multitask



Dual-core systems enable multitasking operating systems to execute two tasks simultaneously

# Process and Thread?



# More Background Threads

Nothing little app

```
public class ThreadDumpSample {  
    public static void main(String[] args) {  
        Thread.sleep(100000);  
    }  
}
```

C:\Users\nadeen\Desktop>jps -l  
3296 sun.tools.jps.Jps  
4276 com.prokarma.app.gtp.thread.ThreadDumpSample  
C:\Users\nadeen\Desktop>jstack 4276 > ThreadDump.txt

This is how, thread dump is created



## Thread Dump of ThreadDumpSample

```
Full thread dump Java HotSpot(TM) 64-Bit Server VM (20.45-b01 mixed mode):  
"Low Memory Detector" daemon prio=6 tid=0x000000000064b5800 mid=0x1238 runnable [0x0000000000000000]  
    java.lang.Thread.State: RUNNABLE  
"C2 CompilerThread1" daemon prio=10 tid=0x000000000064b2000 mid=0x11b0 waiting on condition [0x0000000000000000]  
    java.lang.Thread.State: RUNNABLE  
"C2 CompilerThread0" daemon prio=10 tid=0x0000000000529000 mid=0xe30 waiting on condition [0x0000000000000000]  
    java.lang.Thread.State: RUNNABLE  
"Attach Listener" daemon prio=10 tid=0x0000000000527800 mid=0xdabc waiting on condition [0x0000000000000000]  
    java.lang.Thread.State: RUNNABLE  
"Signal Dispatcher" daemon prio=10 tid=0x000000000064a0800 mid=0x81c runnable [0x0000000000000000]  
    java.lang.Thread.State: RUNNABLE  
"Finalizer" daemon prio=8 tid=0x0000000000512800 mid=0x804 in Object.wait() [0x0000000000645f000]  
    java.lang.Thread.State: WAITING (on object monitor)  
    at java.lang.Object.wait(Native Method)  
    - waiting on <0x00000007d5801300> (a java.lang.ref.ReferenceQueue$Lock)  
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:118)  
    - locked <0x00000007d5801300> (a java.lang.ref.ReferenceQueue$Lock)  
    at java.lang.ref.ReferenceQueue.remove(ReferenceQueue.java:134)  
    at java.lang.ref.Finalizer$FinalizerThread.run(Finalizer.java:171)  
"Reference Handler" daemon prio=10 tid=0x0000000000509800 mid=0xf30 in Object.wait() [0x000000000636f000]  
    java.lang.Thread.State: WAITING (on object monitor)  
    at java.lang.Object.wait(Native Method)  
    - waiting on <0x00000007d58011d8> (a java.lang.ref.Reference$Lock)  
    at java.lang.Object.wait(Object.java:485)  
    at java.lang.ref.Reference$ReferenceHandler.run(Reference.java:116)  
    - locked <0x00000007d58011d8> (a java.lang.ref.Reference$Lock)  
"main" prio=6 tid=0x000000000060b000 mid=0x1368 waiting on condition [0x000000000254f000]  
    java.lang.Thread.State: TIMED_WAITING (sleeping)  
    at java.lang.Thread.sleep(Native Method)  
    at com.prokarma.app.gtp.thread.ThreadDumpSample.main(ThreadDumpSample.java:6)  
"VM Thread" prio=10 tid=0x0000000000501000 mid=0x13e4 runnable  
"GC task thread#0 (ParallelGC)" prio=6 tid=0x000000000045f000 mid=0x1104 runnable  
"GC task thread#1 (ParallelGC)" prio=6 tid=0x0000000000460800 mid=0x4a4 runnable  
"VM Periodic Task Thread" prio=10 tid=0x00000000064cc000 mid=0x1398 waiting on condition  
JNI global references: 883
```

# Basic Operations on Thread

```
public class BasicThreadOperations {  
  
    public static void main(String[] args) {  
  
        // Get Control of the current thread  
        Thread currentThread = Thread.currentThread();  
        //Print Thread information  
        System.out.println("Thread Name : " + currentThread.getName());  
        System.out.println("Thread Group : " + currentThread.getThreadGroup());  
        System.out.println("Thread Priority : " + currentThread.getPriority());  
        System.out.println("Thread is Daemon : " + currentThread.isDaemon());  
        System.out.println("Thread State : " + currentThread.getState());  
        //Update current thread details  
        currentThread.setName("The Main Thread");  
        currentThread.setPriority(6);  
  
        System.out.println("\nNew Thread Name : " + currentThread.getName());  
        System.out.println("New Thread Priority : " + currentThread.getPriority());  
        System.out.println();  
  
        for (int i = 0; i < 6; i++) {  
            System.out.println("Current value of i = " + i);  
            System.out.println("Going to Sleep for 1 second");  
            try {  
                Thread.sleep(1000);  
            } catch (InterruptedException e) {  
                System.out.println("Somebody interrupted " + currentThread + " for un  
            }  
            System.out.println("Woke up");  
        }  
        System.out.println("Going to Die...don't stop me.");  
    }  
}
```

## Out put

```
Thread Name : main  
Thread Group : java.lang.ThreadGroup[name=main,maxpri=1]  
Thread Priority : 5  
Thread is Daemon : false  
Thread State : RUNNABLE  
  
New Thread Name : The Main Thread  
New Thread Priority : 6  
  
Current value of i = 0  
Going to Sleep for 1 second  
Woke up  
Current value of i = 1  
Going to Sleep for 1 second  
Woke up  
Current value of i = 2  
Going to Sleep for 1 second  
Woke up  
Current value of i = 3  
Going to Sleep for 1 second  
Woke up  
Current value of i = 4  
Going to Sleep for 1 second  
Woke up  
Current value of i = 5  
Going to Sleep for 1 second  
Woke up  
Going to Die...don't stop me.
```



Let's clarify some terms

---

**Concurrency ≠ Parallelism**

**Multithreading ≠ Parallelism**



# Few definitions

---

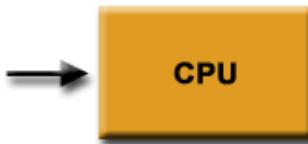
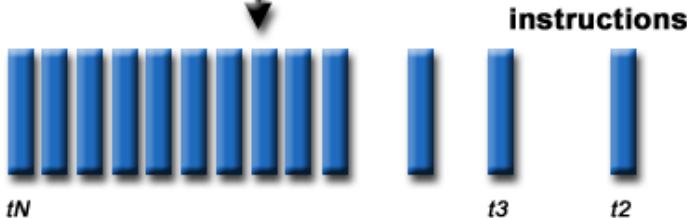
**Concurrency:** A condition that exists when at least two threads are *making progress*

**Parallelism:** A condition that arises when at least two threads are **executing simultaneously**

**Multithreading:** Allows access to two or more threads. Execution occurs in more than one thread of control, using parallel or concurrent processing

*Source: Sun's Multithreaded Programming Guide*  
<http://dlc.sun.com/pdf/816-5137/816-5137.pdf>

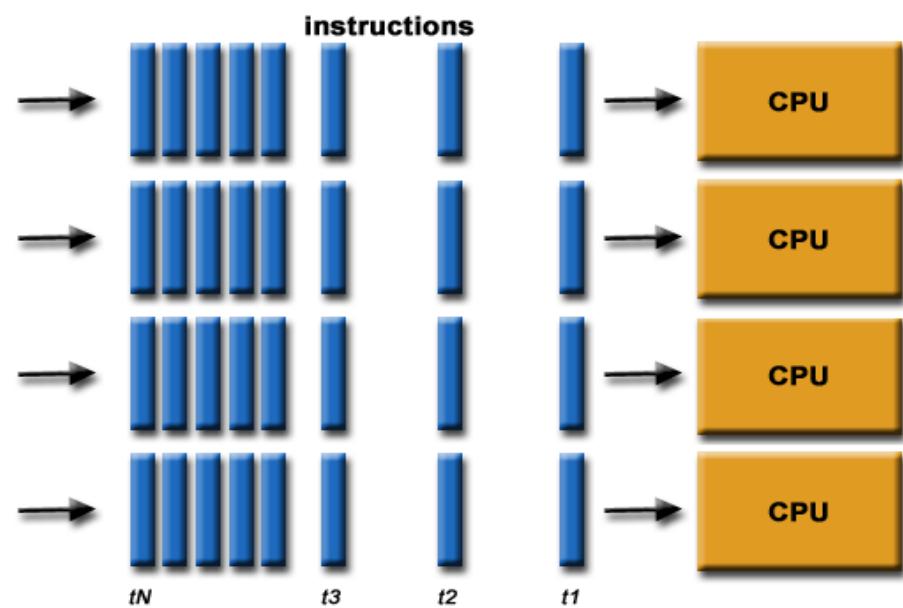
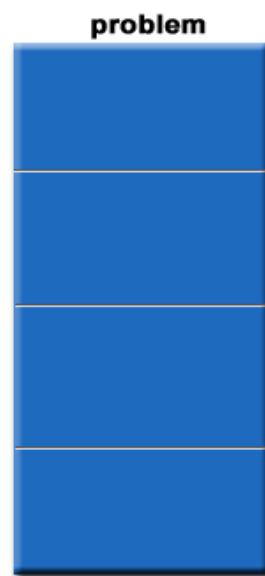




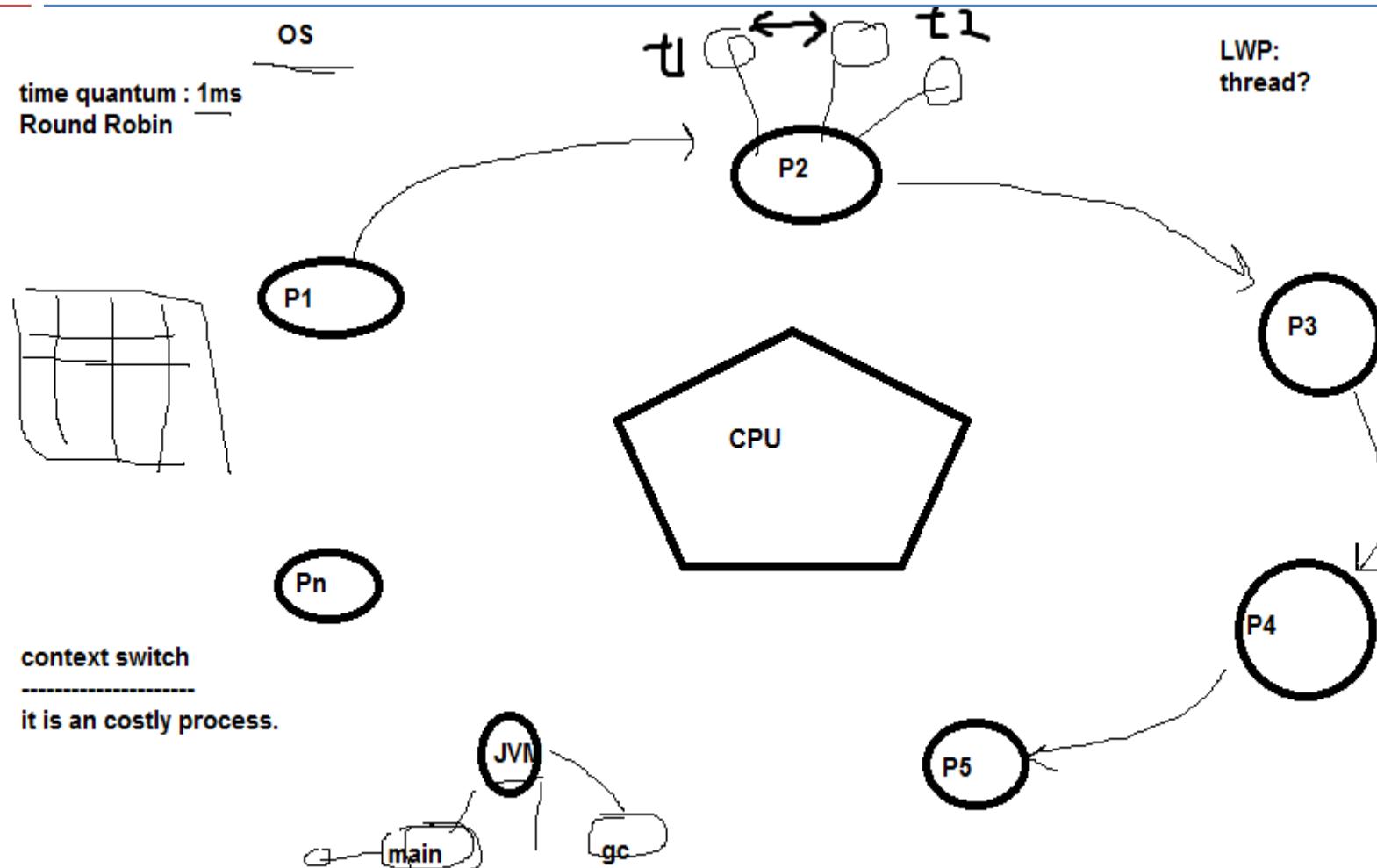
Concurrency

VS

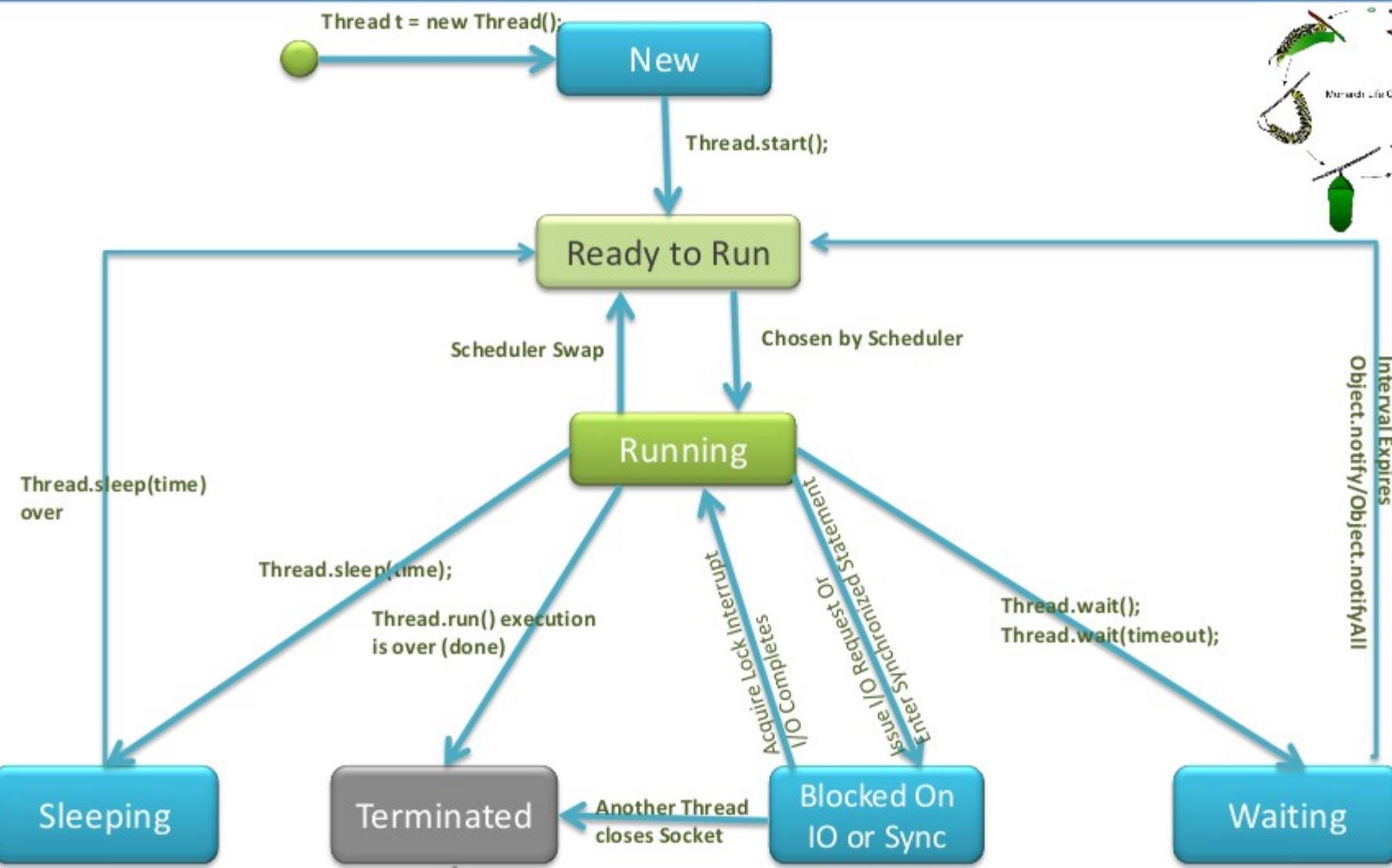
Parallelism



# What is threads? LWP



# Thread Lifecycle



# Important methods of Thread

- **start()**
  - Makes the Thread Ready to run
- **isAlive()**
  - A Thread is alive if it has been started and not died.
- **sleep(milliseconds)**
  - Sleep for number of milliseconds.
- **isInterrupted()**
  - Tests whether this thread has been interrupted.
- **Interrupt()**
  - Indicate to a Thread that we want to finish. If the thread is blocked in a method that responds to interrupts, an InterruptedException will be thrown in the other thread, otherwise the interrupt status is set.
- **join()**
  - Wait for this thread to die.
- **yield()**
  - Causes the currently executing thread object to temporarily pause and allow other threads to execute.

## sleep() yield() wait()

- sleep(n) says "***I'm done with my time slice, and please don't give me another one for at least n milliseconds.***" The OS doesn't even try to schedule the sleeping thread until requested time has passed.
- yield() says "***I'm done with my time slice, but I still have work to do.***" The OS is free to immediately give the thread another time slice, or to give some other thread or process the CPU the yielding thread just gave up.
- .wait() says "***I'm done with my time slice. Don't give me another time slice until someone calls notify().***" As with sleep(), the OS won't even try to schedule your task unless someone calls notify() (or one of a few other wakeup scenarios occurs).

# sleep() and wait()

| Object.wait()                                                                                                                                                                                                                                             | Thread.sleep()                                                                  |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------|
| Called from Synchronized block                                                                                                                                                                                                                            | No Such requirement                                                             |
| Monitor is released                                                                                                                                                                                                                                       | Monitor is not released                                                         |
| Awake when notify() and notifyAll() method is called on the monitor which is being waited on.                                                                                                                                                             | Not awake when notify() or notifyAll() method is called, it can be interrupted. |
| not a static method                                                                                                                                                                                                                                       | static method                                                                   |
| wait() is generally used on condition                                                                                                                                                                                                                     | sleep() method is simply used to put your thread on sleep.                      |
| Can get <i>spurious wakeups</i> from wait (i.e. the thread which is waiting resumes for no apparent reason). We Should always wait whilst spinning on some condition , ex :<br><pre>synchronized {<br/>    while (!condition) monitor.wait();<br/>}</pre> | This is deterministic.                                                          |
| Releases the lock on the object that wait() is called on                                                                                                                                                                                                  | Thread does <i>not</i> release the locks it holds                               |

# Volatile

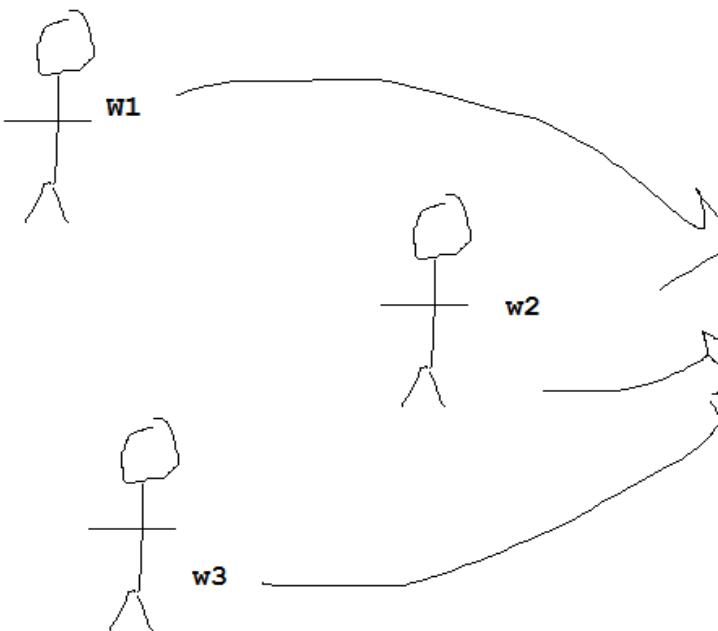
- What is the expected output?
- What is the problem here?

```
public class RaceCondition {  
    private static boolean done;  
  
    public static void main(String[] args) throws InterruptedException {  
        new Thread(new Runnable() {  
            public void run() {  
                int i = 0;  
                while(!done) { i++; }  
                System.out.println("Done! [" + Thread.currentThread().getName() + "]");  
            }  
        }).start();  
  
        TimeUnit.SECONDS.sleep(1);  
        done = true;  
        System.out.println("flag done set to true [" + Thread.currentThread().getName() + "]");  
    }  
}
```

- The program just prints (in windows 7 x64 bit)  
"flag done set to true [main]" and stuck for ever.

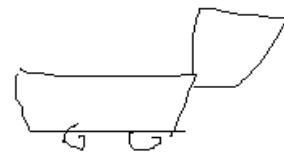
# Creating threads Java?

- Implements Runnable interface
- Extending Thread class.....
- Job and Worker analogy...



**Thread**  
-----  
consider object of threads as  
worker

Implementation of Runnable as  
Job



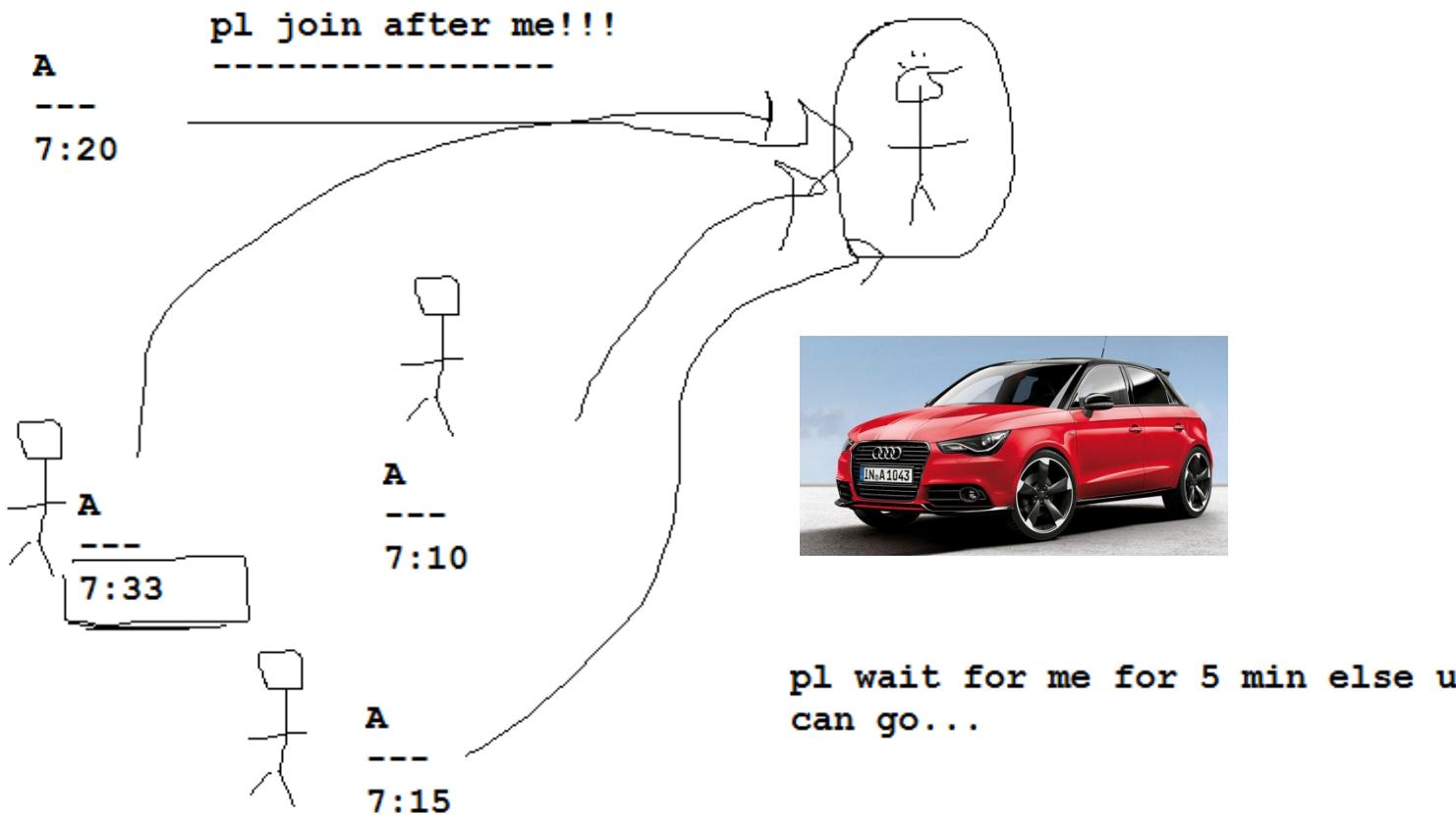
**Job**

**simulation**  
-----  
**sleep()**

```
class Job implements Runnable{  
  
    public void run() {  
        // TODO Auto-generated method stub  
    }  
  
}
```

```
class MyThread extends Thread{  
    public void run() {  
        // TODO Auto-generated method stub  
    }  
}
```

# Understanding join() method



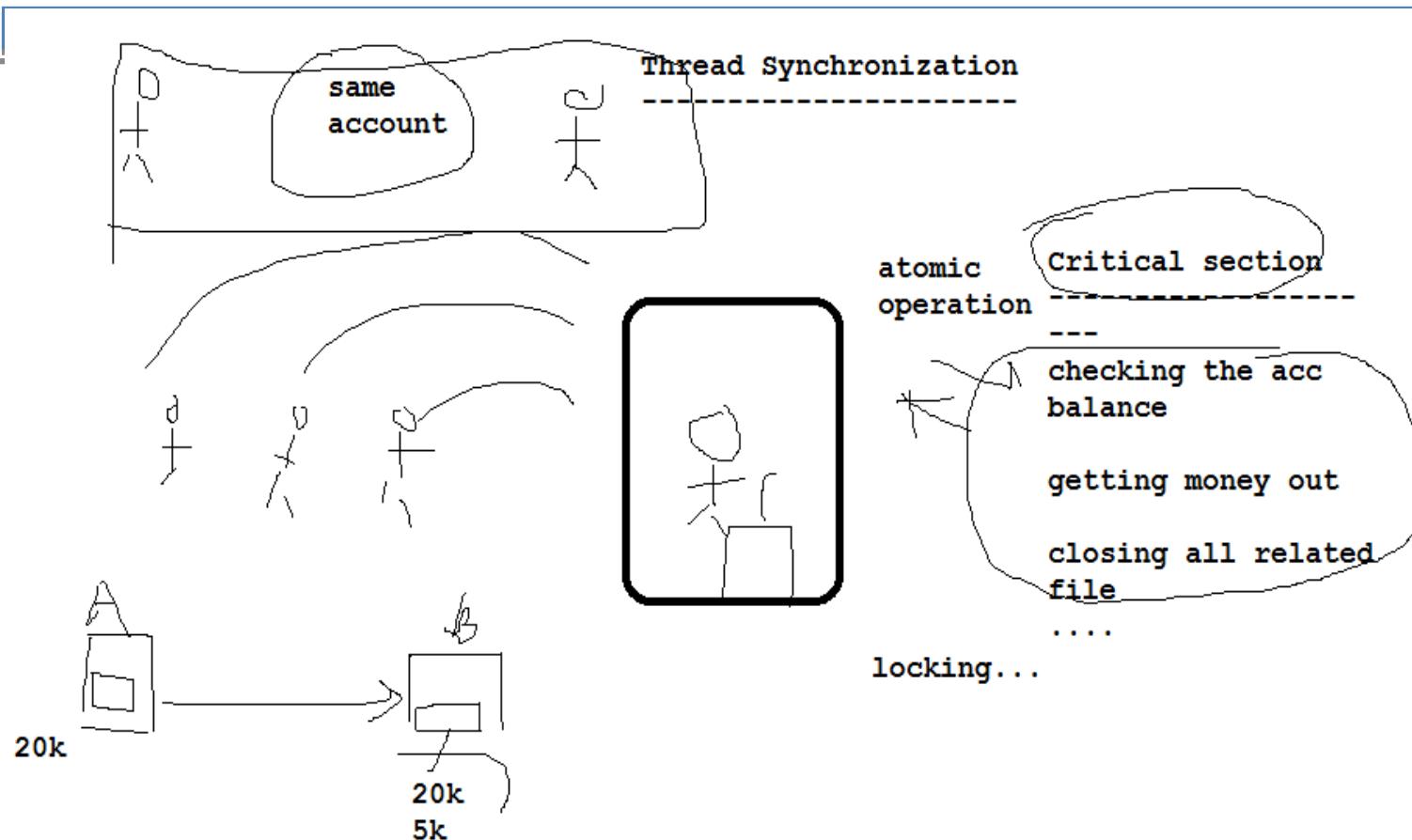
# Checking thread priorities

```
class Clicker implements Runnable
{
    int click=0;
    Thread t;
    private volatile boolean running=true;
    public Clicker(int p)
    {
        t=new Thread(this);
        t.setPriority(p);
    }
    public void run()
    {
        while(running)
            click++;
    }
    public void stop()
    {
        running=false;
    }
    public void start()
    {
        t.start();
    }
}
```

```
.....
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
Clicker hi=new Clicker(Thread.NORM_PRIORITY+2);
Clicker lo=new Clicker(Thread.NORM_PRIORITY-2);
lo.start();
hi.start();
try
{
    Thread.sleep(10000);
}
catch(InterruptedException ex){}
lo.stop();
hi.stop();
//wait for child to terminate
try
{
    hi.t.join();
    lo.t.join();
}
catch(InterruptedException ex)
{
}

System.out.println("Low priority thread:"+lo.click);
System.out.println("High priority thread:"+hi.click);
.....
....
```

# Understanding thread synchronization



# Synchronization

- Synchronization
  - Mechanism to controls the order in which threads execute
  - Competition vs. cooperative synchronization
- Mutual exclusion of threads
  - Each synchronized method or statement is guarded by an object.
  - When entering a synchronized method or statement, the object will be locked until the method is finished.
  - When the object is locked by another thread, the current thread must wait.

# Synchronized Keyword

- Synchronizing instance method

```
class SpeechSynthesizer {  
    synchronized void say( String words ) {  
        // speak  
    }  
}
```

- Synchronizing multiple methods.

```
class Spreadsheet {  
    int cellA1, cellA2, cellA3;  
  
    synchronized int sumRow() {  
        return cellA1 + cellA2 + cellA3;  
    }  
  
    synchronized void setRow( int a1, int a2, int a3 ) {  
        cellA1 = a1;  
        cellA2 = a2;  
        cellA3 = a3;  
    }  
    ...  
}
```

- Synchronizing a block of code.

```
synchronized ( myObject ) {  
    // Functionality that needs exclusive access to resources  
}
```

```
synchronized void myMethod () {  
    ...  
}
```

is equivalent to:

```
void myMethod () {  
    synchronized ( this ) {  
        ...  
    }  
}
```

# Synchronized Keyword

- Marking a method as synchronized

```
public class Incrementor {  
    private int value = 0;  
    public synchronized int increment() {  
        return ++value;  
    }  
}
```

- Using explicit lock object

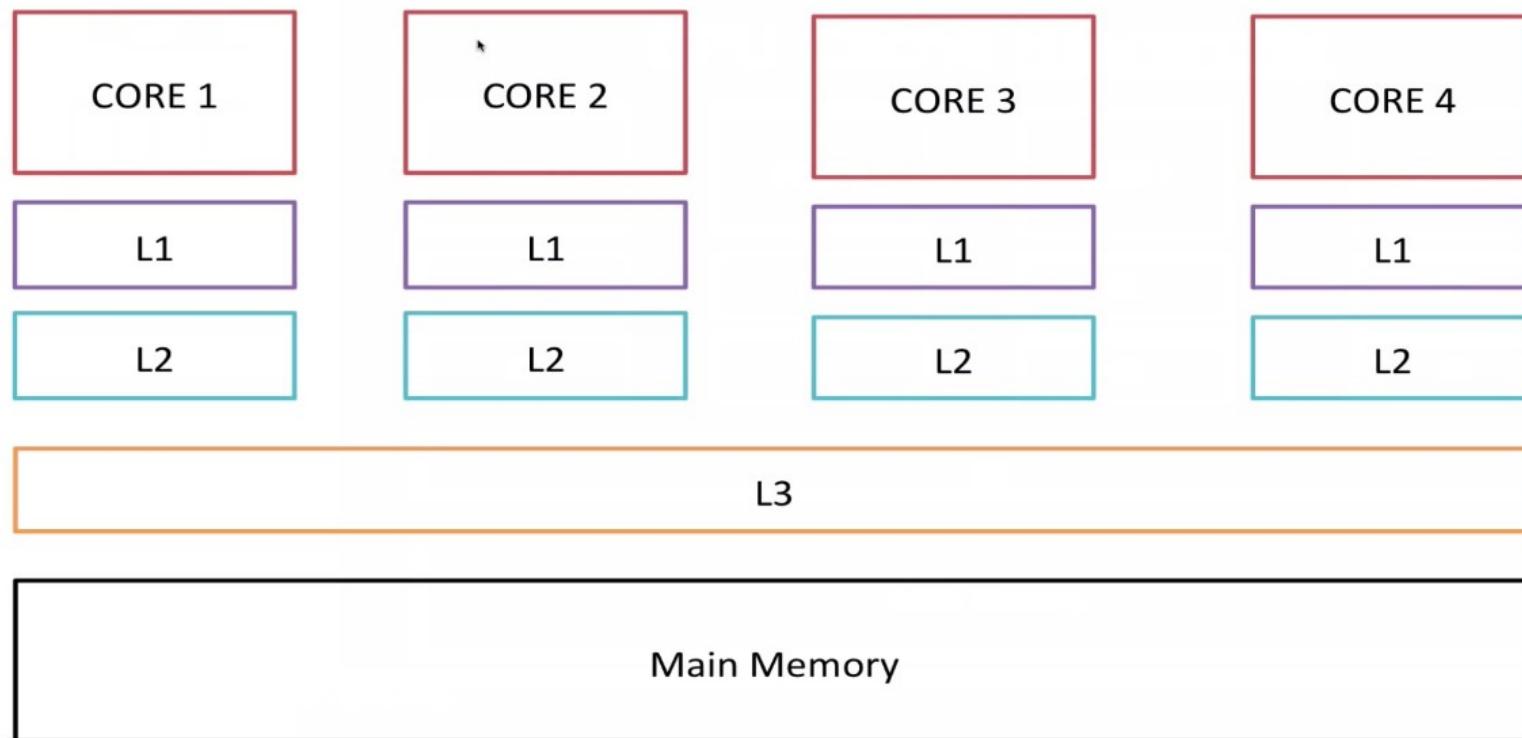
```
public class Incrementor {  
    private final Object lock = new Object();  
    private int value = 0;  
    public int increment() {  
        synchronized (lock) {  
            return ++value;  
        }  
    }  
}
```

- Using the this Monitor

```
public class Incrementor {  
    private int value = 0;  
    public int increment() {  
        synchronized (this) {  
            return ++value;  
        }  
    }  
}
```

# volatile

## CPU Cache Hierarchy



# Using thread synchronization

```
class CallMe
{
    synchronized void call(String msg)
    {
        System.out.print("[ "+msg);
        try
        {
            Thread.sleep(500);
        }
        catch(InterruptedException ex) {}
        System.out.println(" ]");
    }
}
```

```
class Caller implements Runnable
{
    String msg;
    CallMe target;
    Thread t;
    public Caller(CallMe targ,String s)
    {
        target=targ;
        msg=s;
        t=new Thread(this);
        t.start();
    }
    public void run()
    {
        target.call(msg);
    }
}
```

```
Caller ob1=new Caller(target, "Hello");
Caller ob2=new Caller(target,"Synchronized");
Caller ob3=new Caller(target,"Java");
```

# Inter thread communication

- Java have elegant Interprocess communication using `wait()` `notify()` and `notifyAll()` methods
- All these method defined final in the Object class
- Can be only called from a synchronized context

# wait() and notify(), notifyAll()

## ▫ wait()

- Tells the calling thread to give up the monitor and go to the sleep until some other thread enter the same monitor and call notify()

## ▫ notify()

- Wakes up the first thread that called wait() on same object

## ▫ notifyAll()

- Wakes up all the thread that called wait() on same object, highest priority thread is going to run first

# Incorrect implementation of

```
class Q
{
    int n;
    synchronized int get()
    {
        System.out.println("got:"+n);
        return n;
    }
    synchronized void put(int n)
    {
        this.n=n;
        System.out.println("Put:"+n);
    }
}
```

```
public class PandC {
public static void main(String[] args) {
    Q q=new Q();
    new Producer(q);
    new Consumer(q);
    System.out.println("ctrl C for exit");
}
}
```

S

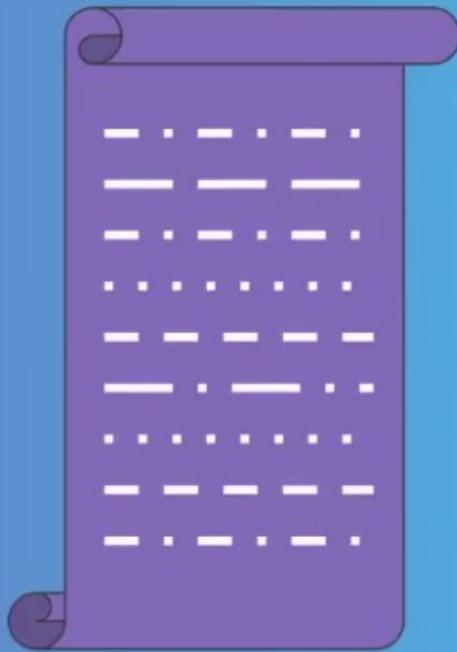
```
class Producer implements Runnable
{
    Q q;
    public Producer(Q q) {
        this.q=q;
        new Thread(this,"Producer").start();
    }
    public void run()
    {
        int i=0;
        while(true)
            q.put(i++);
    }
}
```

```
class Consumer implements Runnable
{
    Q q;
    Consumer(Q q)
    {
        this.q=q;
        new Thread(this,"consumer").start();
    }
    public void run()
    {
        while(true)
            q.get();
    }
}
```

# Correct implementation of producer consumer ...

```
class Q
{   int n;
    boolean valueSet=false;
    synchronized int get()
    {
        if(!valueSet)
            try
            {
                wait();
            }
        catch(InterruptedException ex){}
        System.out.println("got:"+n);
        valueSet=false;
        notify();
        return n;
    }
    synchronized void put(int n)
    {
        if(valueSet)
            try
            {
                wait();
            }
        catch(InterruptedException ex){}
        this.n=n;
        valueSet=true;
        System.out.println("Put:"+n);
        notify();
    }
}
```

# Java Memory Model



## What is Java Memory Model?

*JMM is a specification which guarantees visibility of fields (aka happens before) amidst reordering of instructions.*

# Out of order execution

Performance driven changes  
done by  
Compiler, JVM or CPU

## Instructions

```
a = 3;  
b = 2;  
a = a + 1;
```

→ Load a  
→ Set to 3  
→ Store a

→ Load b  
→ Set to 2  
→ Store b

→ Load a  
→ Set to 4  
→ Store a

Consider this code

## Instructions

a = 3;

a = a + 1;

b = 2;

- Load a
- Set to 3
- Set to 4
- Store a

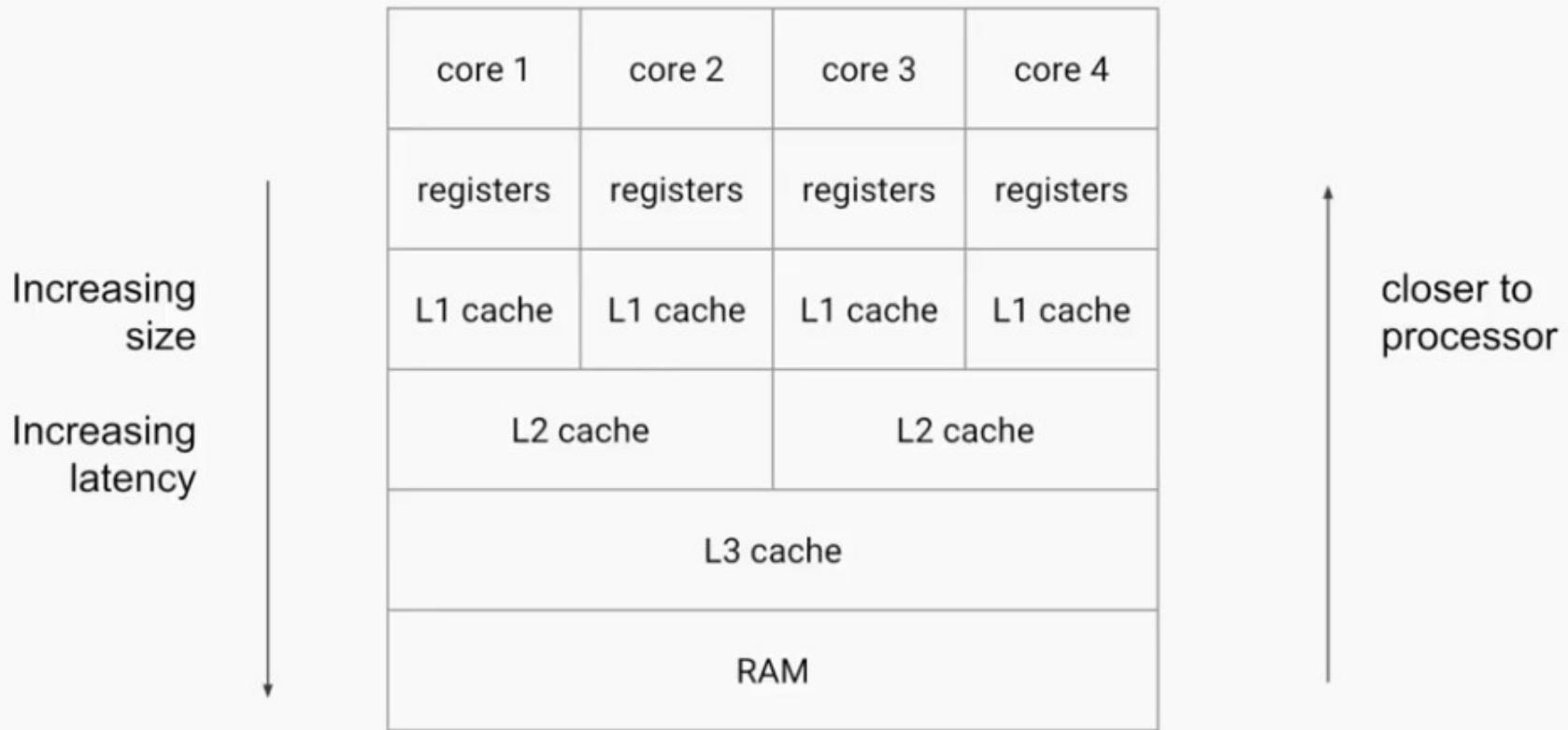
- Load b
- Set to 2
- Store b

# Field Visibility

In presence of multiple threads

a.k.a

Concurrency



```
public class FieldVisibility {  
  
    int x = 0;  
  
    public void writerThread() {  
        x = 1;  
    }  
  
    public void readerThread() {  
        int r2 = x;  
    }  
}
```

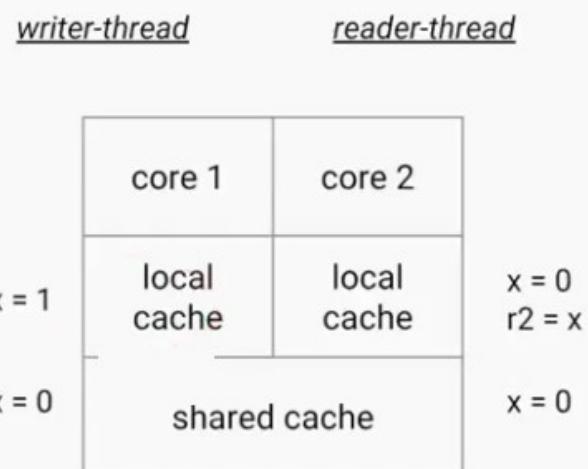
*writer-thread*                    *reader-thread*

|              |             |
|--------------|-------------|
| core 1       | core 2      |
| local cache  | local cache |
| shared cache |             |

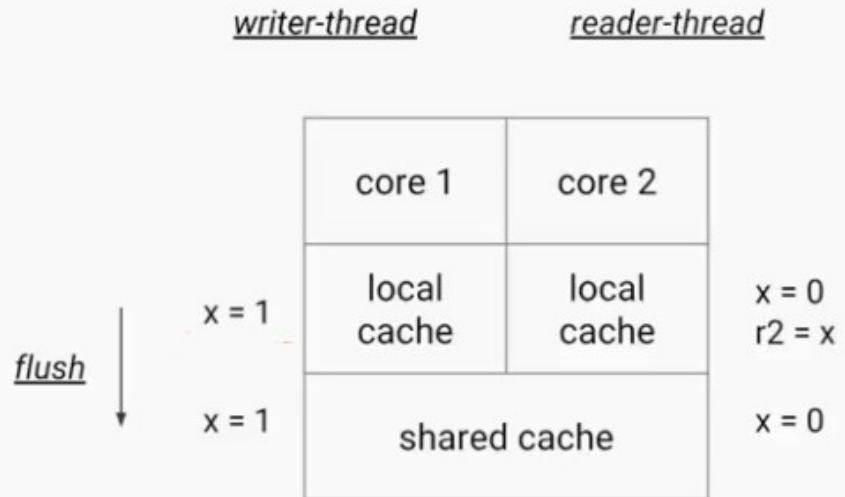
x = 0

x = 0

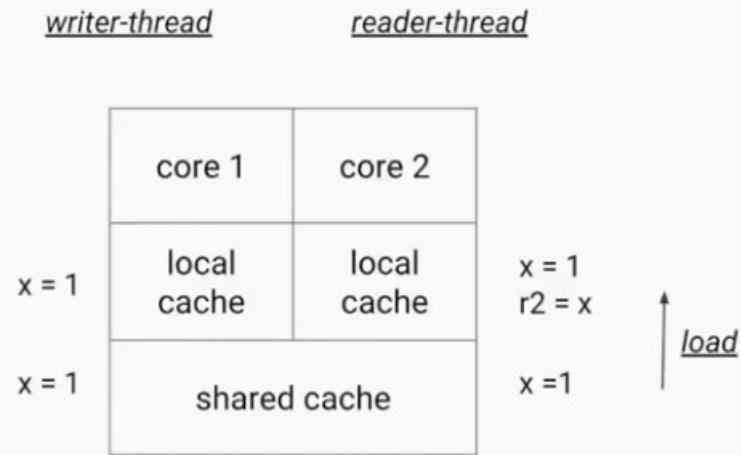
```
public class FieldVisibility {  
  
    int x = 0;  
  
    public void writerThread() {  
        x = 1;  
    }  
  
    public void readerThread() {  
        int r2 = x;  
    }  
}
```



```
public class VolatileVisibility {  
  
    volatile int x = 0;  
  
    public void writerThread() {  
        x = 1;  
    }  
  
    public void readerThread() {  
        int r2 = x;  
    }  
}
```



```
public class VolatileVisibility {  
    volatile int x = 0;  
    •  
    public void writerThread() {  
        x = 1;  
    }  
    public void readerThread() {  
        int r2 = x;  
    }  
}
```



```
public class VolatileFieldsVisibility {

    int a = 0, b = 0, c = 0;
    volatile int x = 0;

    public void writerThread() {

        a = 1;
        b = 1;
        c = 1;

        x = 1; // write of x
    }

    public void readerThread() {

        int r2 = x; // read of x

        int d1 = a;
        int d2 = b;
        int d3 = c;
    }
}
```

JMM rule:  
Whatever happens before `x=1`  
in writer thread  
must be visible in reader thread  
after `int r2=x;`

"Happens-before" relationship for volatile

# Not only volatile

Also

1. Synchronized
2. Locks
3. Concurrent collections
4. Thread operations (join,start)

final fields (special behavior)

```
public class SynchronizedFieldsVisibility {

    int a = 0, b = 0, c = 0;
    volatile int x = 0;

    public void writerThread() {
        a = 1;
        b = 1;
        c = 1;

        synchronized (this) {
            x = 1;
        }
    }

    public void readerThread() {
        synchronized (this) {
            int r2 = x;
        }

        int d1 = a;
        int d2 = b;
        int d3 = c;
    }
}
```

Same behavior with synchronized blocks. V.Imp: Has to be synchronized on same object!!

```
public class SynchronizedFieldsVisibility {

    int a = 0, b = 0, c = 0;
    volatile int x = 0;

    public void writerThread() {

        synchronized (this) {
            a = 1;
            b = 1;
            c = 1;
            x = 1;
        }
    }

    public void readerThread() {

        synchronized (this) {
            int r2 = x;
            int d1 = a;
            int d2 = b;
            int d3 = c;
        }
    }
}
```

Better to be more clear though!

```
public class LockVisibility {

    int a = 0, b = 0, c = 0, x = 0;
    Lock lock = new ReentrantLock();

    public void writerThread() {

        lock.lock();
        a = 1;
        b = 1;
        c = 1;
        x = 1;
        lock.unlock();
    }

    public void readerThread() {

        lock.lock();
        int r2 = x;
        int d1 = a;
        int d2 = b;
        int d3 = c;
        lock.unlock();
    }
}
```

```
public class VolatileVisibility {  
  
    boolean flag = true;  
  
    public void writerThread() {  
        flag = false;  
    }  
  
    public void readerThread() {  
        while (flag) {  
            // do some operations  
        }  
    }  
}
```

wrong

```
public class VolatileVisibility {  
  
    volatile boolean flag = true;  
  
    public void writerThread() {  
        flag = false;  
    }  
  
    public void readerThread() {  
        while (flag) {  
            // do some operations  
        }  
    }  
}
```

right

Code on the left can have reader thread keep running. Fixed with volatile flag.

# CONCURRENCY

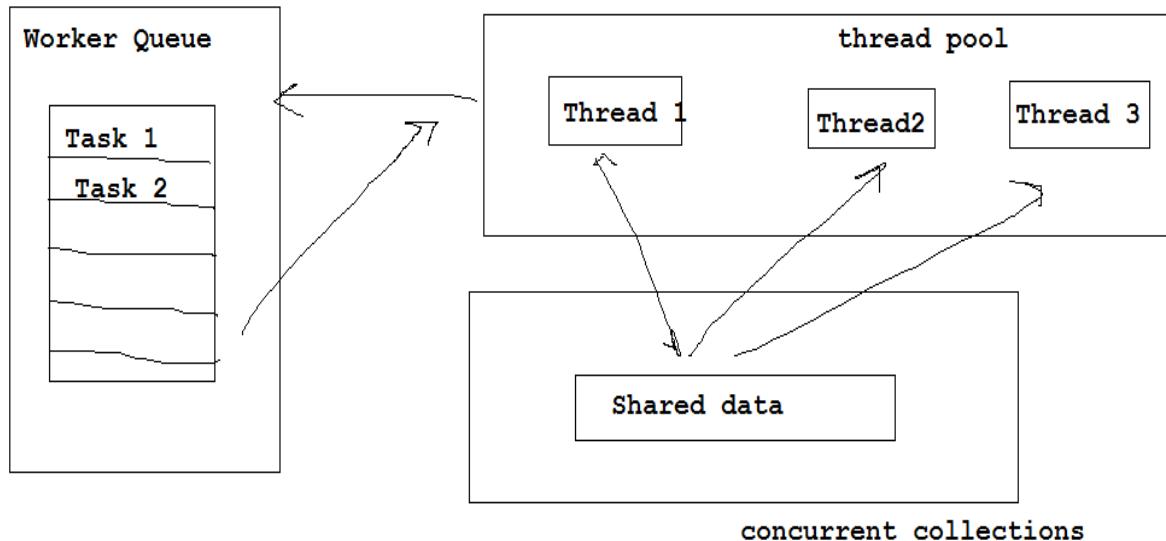
# java.util.concurrency j.u.c

## □ Introduction

- Concurrency : why very active now?
- Building block of concurrent application
- Java concurrency -over the year
- Fork join -introduction
- Coding examples....

# Building block of concurrent application

Building block of concurrent applications



- Thread pool
- Shared data
- Worker queue

# Java concurrency -Pre Java 5

- Task Executor
  - No such framework
  - Runnable/Thread
  - No of task= no of threads
  - Use wait() and notify() for thread intercommunication
- Collection
  - Vector, Hashtable
- Worker queue
  - No framework for task dispatching
  - Create an thread when a task to run
- Too primitive low level
  - High cost of thread management
  - Synchronization=poor performance
  - <<Runnable>> can not return values

# Java concurrency -Java 5

- Task Executor
  - Executor Service Framework
  - Support scheduling (Timed execution)
  - Thread pools (lifecycle management)
  - Specify order of execution
  - Task
    - Object of <<Callable>> or <<Runnable>>
    - <<Callable>> support for asynchronous communication call() and return values
    - Future: hold result of processing
- Collections
  - ConcurrentHashMap, CopyOnWriteList
- Worker queue
  - Blocking Queue, Deque.....
- Issues
  - No of task can't altered at run time
  - Very difficult and performance intensive when applied to recursive style of problems.....

# Java concurrency -Java 7

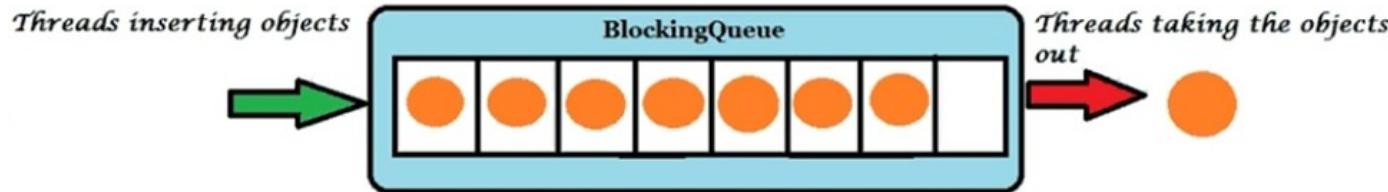
- Fork join framework
  - For supporting a style of programming in which problem are solved recursively splitting them into subtask that are solved in parallel, waiting for them to complete and then composing results !
- JSR 166 Java 7
  - Extend Executor framework for recursive style of problems
- Implements Work stealing algorithm
  - Idle worker “steal” the works from worker who are busy..

# Blocking Queue: introduction

- BlockingQueue is an interface under juc package. It help to contain objects when few threads is inserting object and other threads are taking the object out of it
- 
- The threads will keep on inserting the objects until the max capacity of BlockingQueue is reached, after which the threads will be blocked and they will not be able to insert further objects, threads will be in blocked state until few objects are taken out from BlockingQueue by other threads and there is space available in the blockingqueue
- 
- The threads will keep on taking the objects out from the blockingqueue until there is no object left in blockingqueue, after which the threads will be blocked and they will remain in blocked state until few objects are inserted in the blockingqueue by other threads
- 
- Use put() and take() to implement P and C

The **poll()** doesn't wait if the `messageQueue` is empty. It doesn't care. It just gets the value even if the `messageQueue` is empty. Finally the application ended.

The **take()** method waits at that particular line if the `messageQueue` is empty. It only goes to next line if the `messageQueue` got something inside to process. So **take()** method block the thread and keeps it, which makes the application runs endlessly.



# Blocking Queue: Thread safe DS

---

- `BlockingQueue<E>` is an interface with the following methods

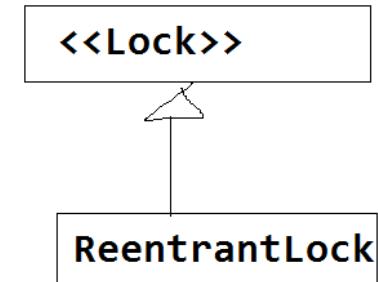
|         | <i>Throws exception</i> | <i>Special value</i>  | <i>Blocks</i>       | <i>Times out</i>                  |
|---------|-------------------------|-----------------------|---------------------|-----------------------------------|
| Insert  | <code>add(e)</code>     | <code>offer(e)</code> | <code>put(e)</code> | <code>offer(e, time, unit)</code> |
| Remove  | <code>remove()</code>   | <code>poll()</code>   | <code>take()</code> | <code>poll(time, unit)</code>     |
| Examine | <code>element()</code>  | <code>peek()</code>   | <i>N/A</i>          | <i>N/A</i>                        |

- until the method returns. If a method *times out* then the method *sblocks* until the time specified is reached, then the method returns.

- Important implementations of `BlockingQueue` are `ArrayBlockingQueue`, `LinkedBlockingQueue`, `PriorityBlockingQueue` and `SynchronousQueue`.

# Synchronization with Locks: ReentrantLock

- Synchronization with Locks
  - More flexible
  - Better performance as compared to wait() notify()



```
class PrintQueue
{
    private final Lock queLock=new ReentrantLock();           ← creating an lock
    public void printJob()
    {
        queLock.lock();                                     ← getting control of
   lock
        try{   ← must be unlock in
            System.out.println("Thread "+Thread.currentThread().getName()+" Done the job");
            Thread.sleep(100);
        }
        catch(InterruptedException ex){}

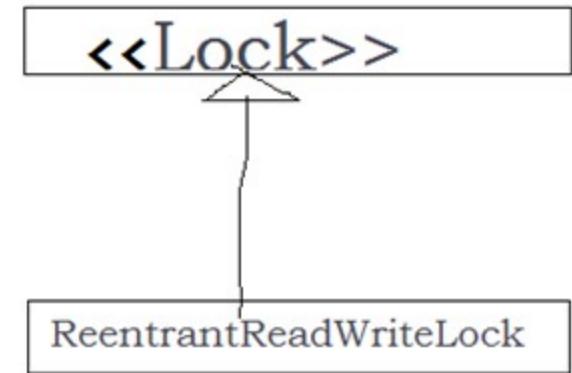
        finally{
            queLock.unlock();                                ← must be unlock in
   finally block
        }
    }
}
```

Annotations in red text are placed near specific code snippets:

- "creating an lock" points to the declaration of the Lock variable.
- "getting control of lock" points to the call to lock().
- "must be unlock in finally block" points to the call to unlock() within the finally block.

# Synchronization with Locks: ReentrantReadWriteLock

- ReentrantReadWriteLock has two locks
  - One for read
    - More than one thread can read
  - One for write
    - Only one thread can write
- Ex: Lets consider case when one thread writing Price Information that is read by more than one reader

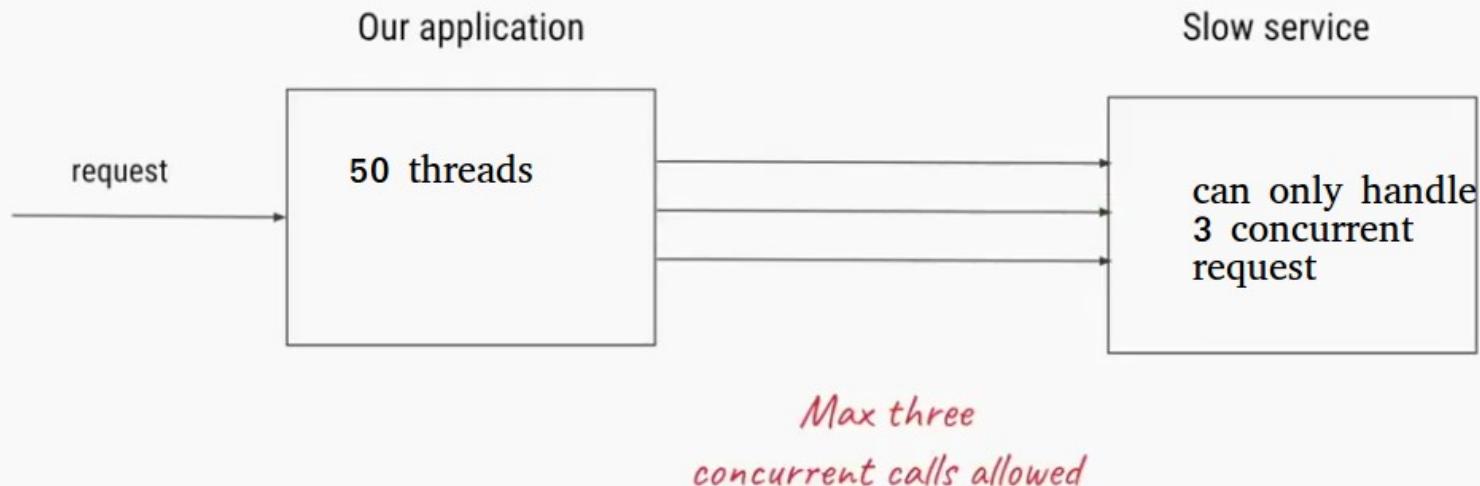


# ReentrantReadWriteLock

```
class PriceInfo
{
    private double price1,price2;
    private ReadWriteLock lock=new ReentrantReadWriteLock();
    PriceInfo(){
        price1=1.0;
        price2=2.0;
    }
    public double getPrice1(){
        lock.readLock().lock();
        double value=price1;
        lock.readLock().unlock();
        return value;
    }
    public double getPrice2(){
        lock.readLock().lock();
        double value=price2;
        lock.readLock().unlock();
        return value;
    }
    public void setPrice(double price1, double price2){
        lock.writeLock().lock();
        this.price1=price1;
        this.price2=price2;
        lock.writeLock().unlock();
    }
}
```

# Semaphores: use cases

Use Case: We have a slow API to class from our application, it can only handle 3 concurrent class while we have 50 thread to call that service? How to handle?

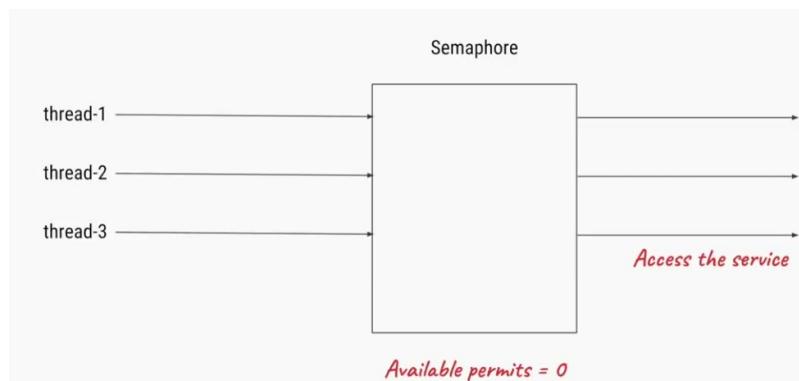
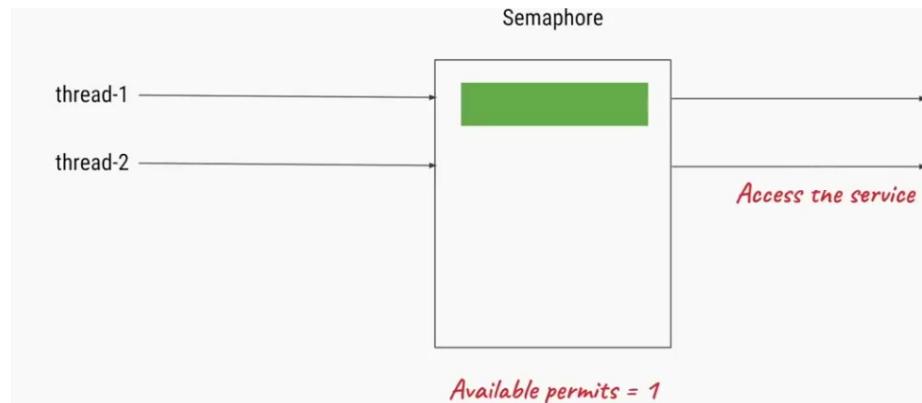


# Semaphores: use cases

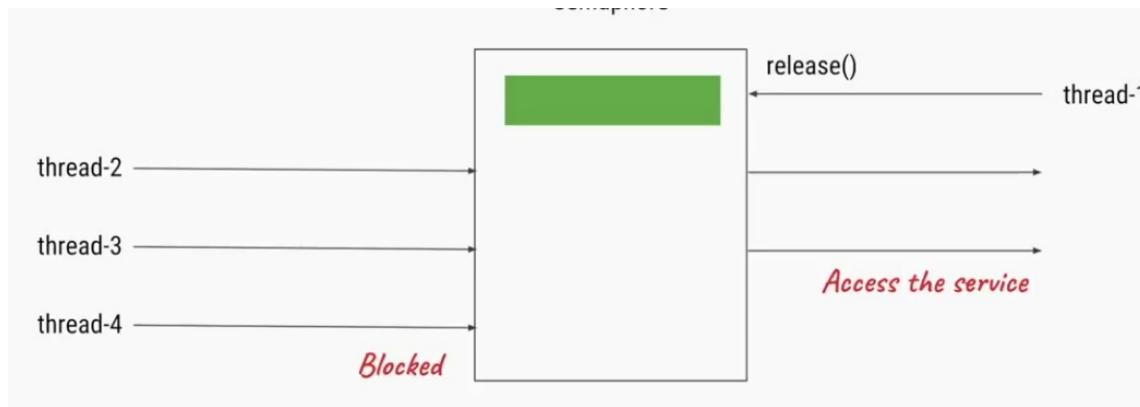
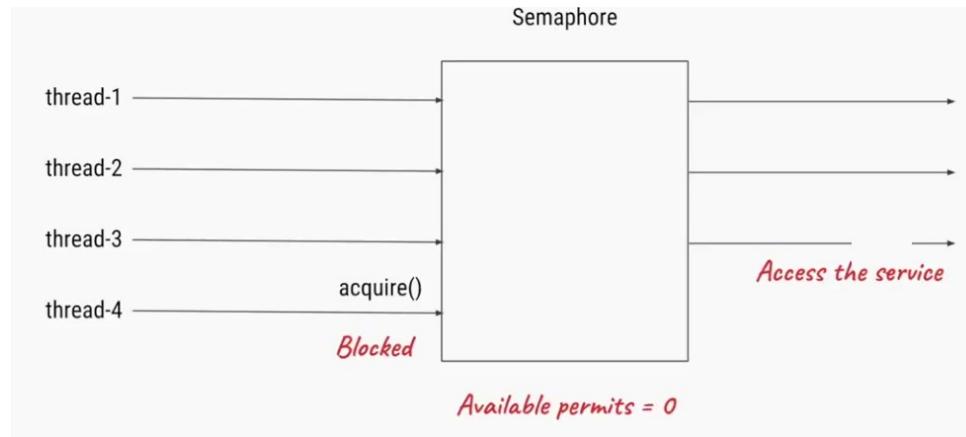
```
public static void main(String[] args) throws InterruptedException {  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 50 );  
    IntStream.of(1000).forEach(i -> service.execute(new Task()));  
  
    service.shutdown();  
    service.awaitTermination( timeout: 1, TimeUnit.MINUTES );  
}  
  
static class Task implements Runnable {  
  
    @Override  
    public void run() {  
        // some processing  
        // IO call to the slow service  
        // rest of processing  
    }  
}
```

← This might be called 50 times concurrently!!

# Semaphores: how permit works?

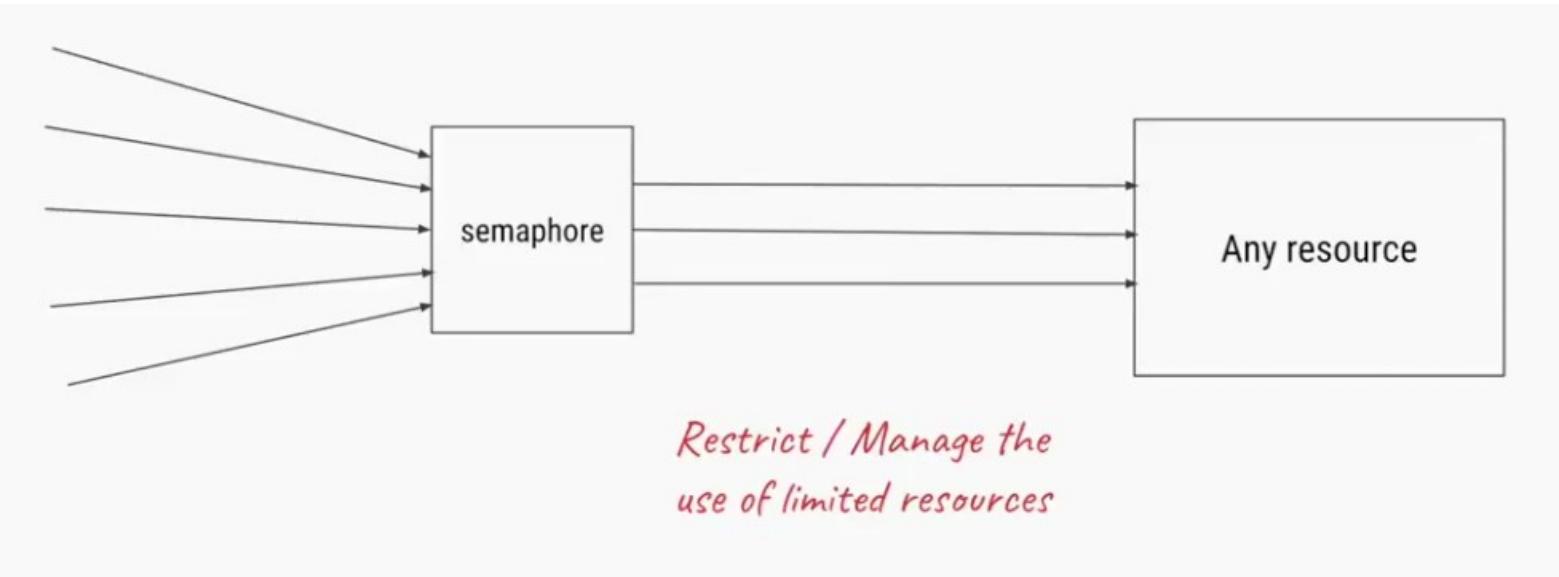


# Semaphores: how permit works?



```
public static void main(String[] args) throws InterruptedException {  
    Semaphore semaphore = new Semaphore( permits: 3);  
  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 50);  
    IntStream.of(1000).forEach(i -> service.execute(new Task(semaphore)));  
  
    service.shutdown();  
    service.awaitTermination( timeout: 1, TimeUnit.MINUTES);  
}  
  
static class Task implements Runnable {  
  
    @Override  
    public void run() {  
        // some processing  
  
        semaphore.acquire(); ← Only 3 threads can acquire  
        // IO call to the slow service  
        semaphore.release();  
        // rest of processing  
    }  
}
```

```
public static void main(String[] args) throws InterruptedException {  
    Semaphore semaphore = new Semaphore( permits: 3);  
  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 50);  
    IntStream.of(1000).forEach(i -> service.execute(new Task(semaphore)));  
  
    service.shutdown();  
    service.awaitTermination( timeout: 1, TimeUnit.MINUTES);  
}  
  
static class Task implements Runnable {  
  
    @Override  
    public void run() {  
        // some processing  
  
        semaphore.acquireUninterruptibly(); Only 3 threads can acquire  
        // IO call to the slow service at a time  
        semaphore.release();  
  
        // rest of processing  
    }  
}
```



| Method                          | Meaning                                                                                 |
|---------------------------------|-----------------------------------------------------------------------------------------|
| tryAcquire                      | Try to acquire, if no permit available, do not block.<br>Continue doing something else. |
| tryAcquire (timeout)            | Same as above but with timeout                                                          |
| availablePermits                | Returns count of permits available                                                      |
| new Semaphore (count, fairness) | FIFO. Fairness guarantee for threads waiting the longest.                               |

# Thread synchronization utilities:CountDownLatch

- Waiting for multiple concurrent events
- Allow one/more thread to wait until a set of operations are completed..
- CountDownLatch class initialized with an integer number i.e. no of operations that thread must wait to complete before they finished
- When a thread want to wait for executions of those operations, it uses await() method, it put thread to sleep until operation are complete
- When one of these operation complete, it use CountDownLatches, countDown() method to decrease the count
- When it (counter ) become zero, the class wake up all the threads that sleeping in await() method
- Ex: CountDownLatch class implementing an video conference system. The system wait for all participant before it begin

# CountDownLatch Ex.

```
class VideoConference implements Runnable
{
    private CountDownLatch controller;

    VideoConference(int n)
    {
        controller=new CountDownLatch(n);
    }

    public void arrive(String name)
    {
        System.out.println("Guest name:"+name+" arrived");
        controller.countDown(); //decrease.
        System.out.println("Controller waiting for more:"+controller.getCount());
    }

    @Override
    public void run() {

        System.out.println("init video conference"+controller.getCount());
        try
        {
            controller.await();
            System.out.println("All comes....lets start now.....");
        }
        catch(InterruptedException ex){}
    }
}
```

```
class Participant implements Runnable
{
    private VideoConference conference;
    private String participantsName;

    Participant(VideoConference conference, String participantsName)
    {
        this.conference=conference;
        this.participantsName=participantsName;
    }
    @Override
    public void run() {
        try
        {
            Thread.sleep(100);
        }
        catch(InterruptedException ex){}
        conference.arrive(participantsName);
    }
}

public class VideoConferenceDemo {

    public static void main(String[] args) {
        VideoConference vc=new VideoConference(10);

        //running conference thread.....
        Thread t=new Thread(vc);
        t.start();

        //now create 10 participants for above conference

        for(int i=0;i<10;i++)
        {
            Participant p=new Participant(vc,"Participant "+i);
            Thread t1=new Thread(p);
            t1.start();
        }
    }
}
```

```
public static void main(String[] args) throws InterruptedException {  
    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4 );  
  
    CountDownLatch latch = new CountDownLatch(3);  
    executor.submit(new DependentService(latch));  
    executor.submit(new DependentService(latch));  
    executor.submit(new DependentService(latch));  
  
    latch.await();  
  
    System.out.println("All dependant services initialized");  
    // program initialized, perform other operations  
}
```

```
public static class DependentService implements Runnable {  
  
    private CountDownLatch latch;  
    public DependentService(CountDownLatch latch) { this.latch = latch; }  
  
    @Override  
    public void run() {  
        // startup task  
        latch.countDown();  
        // continue w/ other operations  
    }  
}
```

### Main thread

### Dependant service threads

Latch count = 3

Latch count = 2

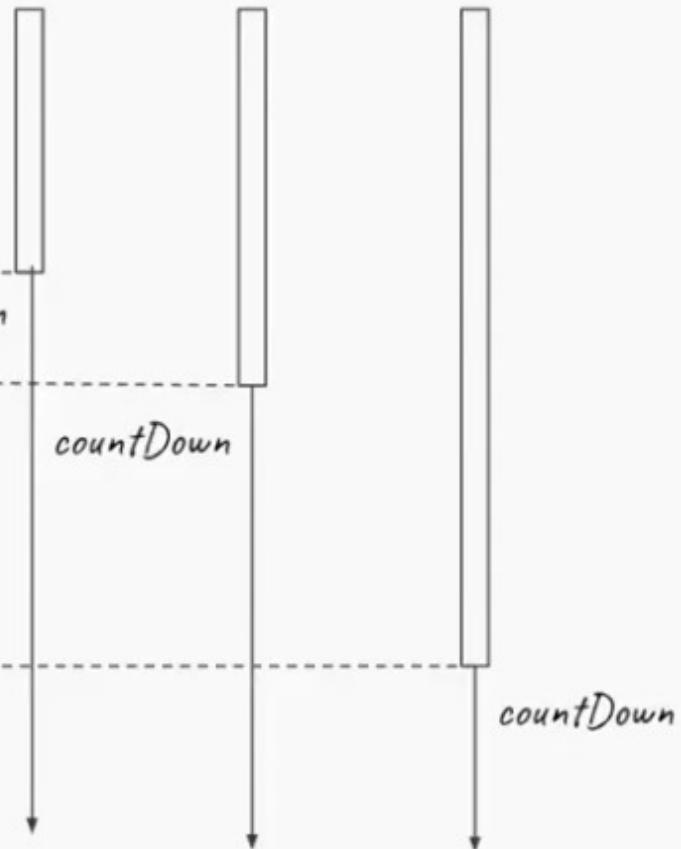
Latch count = 1

Latch count = 0

countDown

countDown

countDown



# CyclicBarrier: Use cases

- Let we have a game of 3 player, we want to repeatedly send message to all 3 player without discrimination, the task run in infine loop ( repedatly), all thread threds say barrier.await() ( all can do in different time) , all three thread need to wait then when all arived then they continue processing

```
public static void main(String[] args) throws InterruptedException {

    ExecutorService executor = Executors.newFixedThreadPool( nThreads: 4);

    CyclicBarrier barrier = new CyclicBarrier( parties: 3);
    executor.submit(new Task(barrier));
    executor.submit(new Task(barrier));
    executor.submit(new Task(barrier));

    Thread.sleep( millis: 2000);
}

public static class Task implements Runnable {

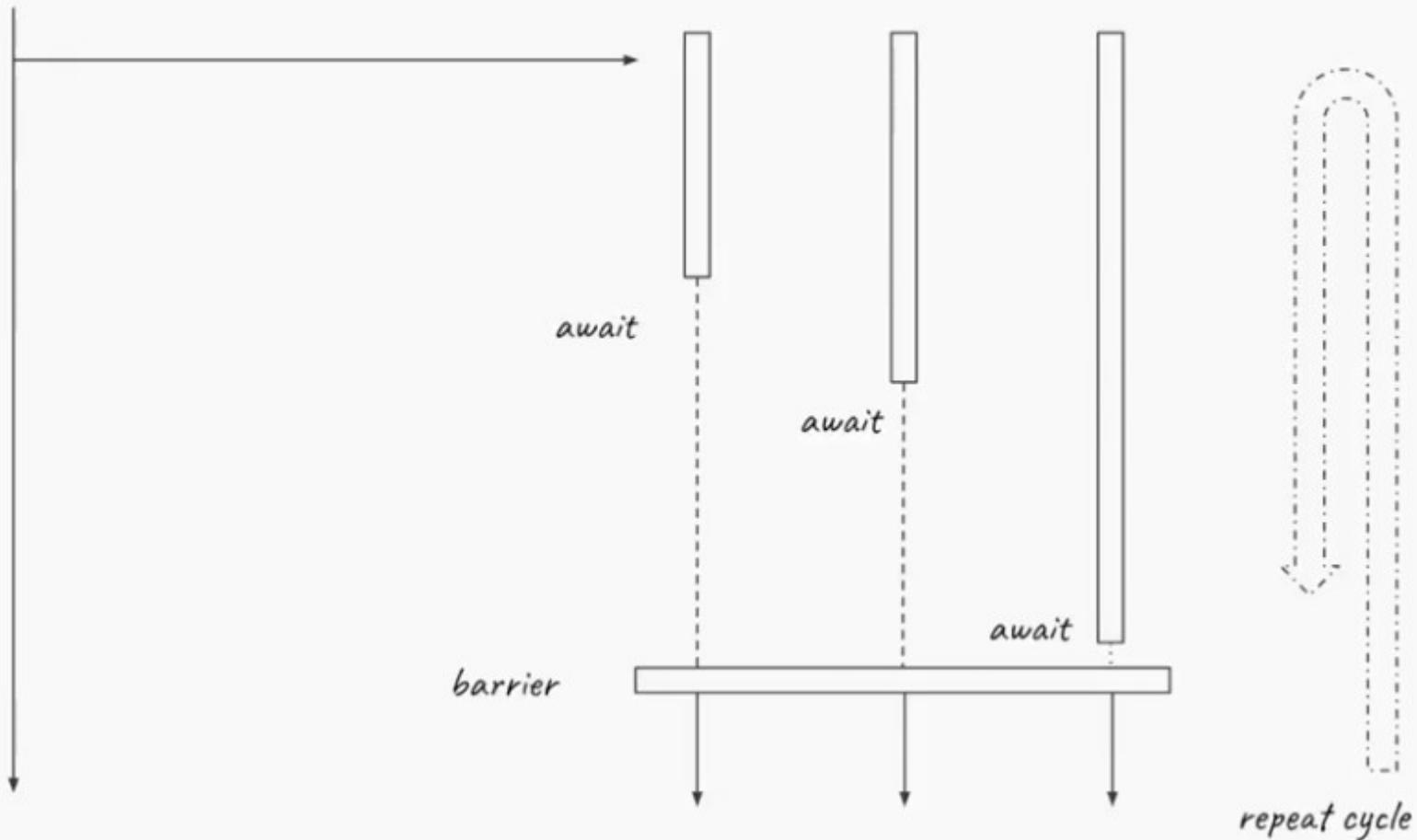
    private CyclicBarrier barrier;
    public Task(CyclicBarrier barrier) { this.barrier = barrier; }

    @Override
    public void run() {

        while (true) {
            try {
                barrier.await();
            } catch (InterruptedException | BrokenBarrierException e) {
                e.printStackTrace();
            }
            // send message to corresponding system
        }
    }
}
```

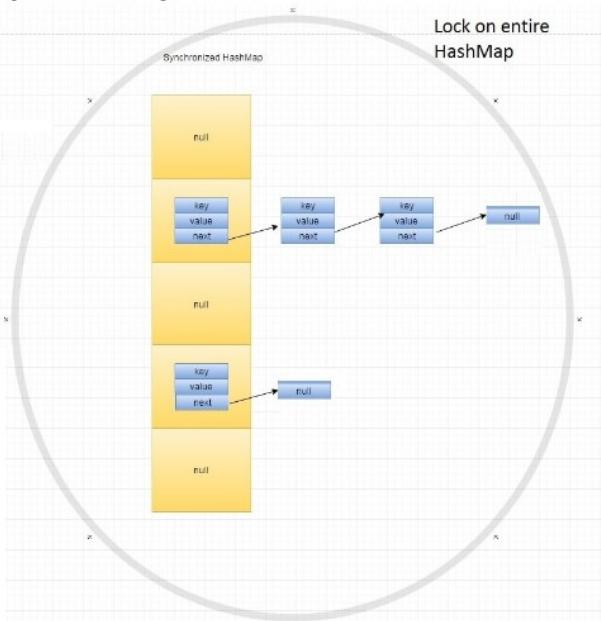
Main thread

Tasks to perform repeatedly



# Thread safe DS java

SynchronizedMap-

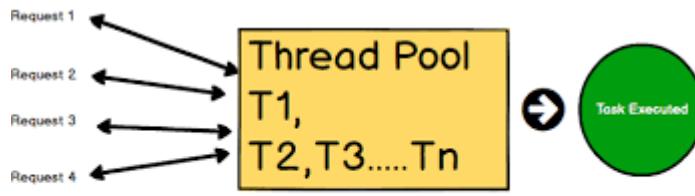


ConcurrentHashMap-



# ExecutorService

Java's thread pool framework



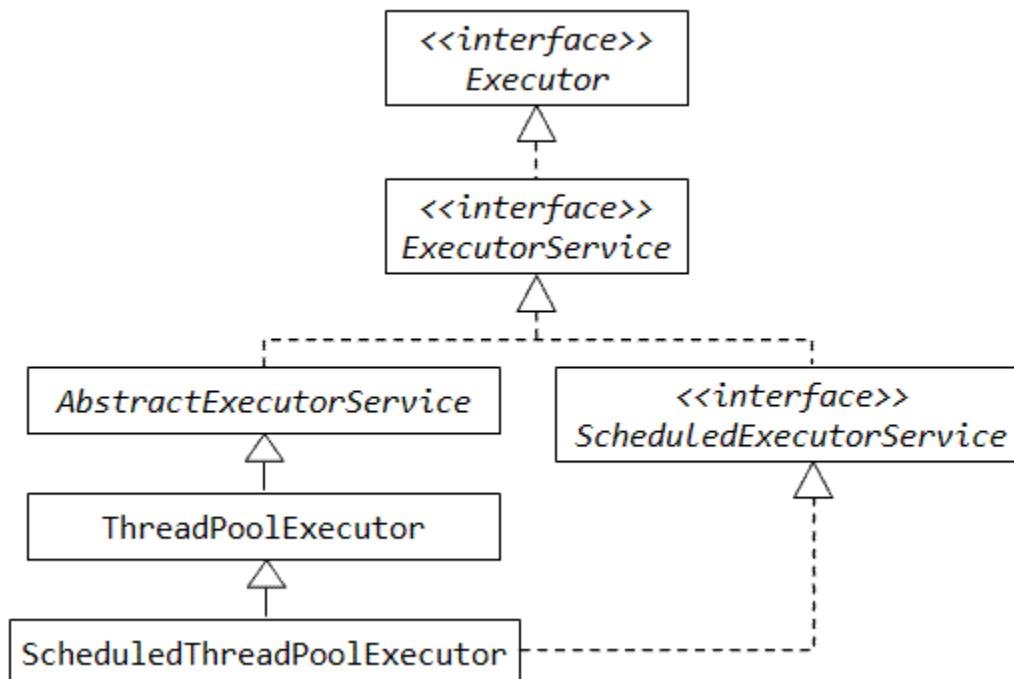
## Thread Pooling

**Thread Pools** are useful when you **need** to limit the number of **threads** running in your application at the same time. ... Instead of starting a new **thread** for every task to execute concurrently, the task can be passed to a **thread pool**. As soon as the **pool** has any idle **threads** the task is assigned to one of them and executed.

Oct 15, 2015

# Executor framework

- It separate task of creation and execution of thread
- We only have to implements Runnable (Make a job) and send it to executor
- Then executor is responsible for execution management
- Executor maintain an thread pool
  - avoid continuously spawning of threads



## Executors, A framework for creating and managing threads. Executors framework helps you with -

- ▶ **Thread Creation:** It provides various methods for creating threads, more specifically a pool of threads, that your application can use to run tasks concurrently.
- ▶ **Thread Management:** It manages the life cycle of the threads in the thread pool. You don't need to worry about whether the threads in the thread pool are active or busy or dead before submitting a task for execution.
- ▶ **Task submission and execution:** Executors framework provides methods for submitting tasks for execution in the thread pool, and also gives you the power to decide when the tasks will be executed. For example, You can submit a task to be executed now or schedule them to be executed later or make them execute periodically.

Java Concurrency API defines the following three executor interfaces that covers everything that is needed for creating and managing threads -

- ▶ **Executor** - A simple interface that contains a method called `execute(Runnable command)` to launch a task specified by a `Runnable` object.
- ▶ **ExecutorService** - A sub-interface of `Executor` that adds functionality to manage the lifecycle of the tasks. It also provides a `submit()` method whose overloaded versions can accept a `Runnable` as well as a `Callable` object. `Callable` objects are similar to `Runnable` except that the task specified by a `Callable` object can also return a value.
- ▶ **ScheduledExecutorService** - A sub-interface of `ExecutorService`. It adds functionality to schedule the execution of the tasks.

# Executor framework Ex With <<Runnable>>

- Consider Task class that contain job to be submitted to executor framework.....

```
class Task implements Runnable
{
    private String name;

    Task(String name){this.name=name;}

    @Override
    public void run() {
        System.out.println("Thread started the job...:"+Thread.currentThread().getName());
        try
        {
            Thread.sleep(100);
        }
        catch(InterruptedException ex){}
        System.out.println("Thread finished the job...:"+Thread.currentThread().getName());
    }
}
```

```
class Server
{
    private ThreadPoolExecutor exe=(ThreadPoolExecutor) Executors.newCachedThreadPool();

    public void executeTask(Task task){
        System.out.println("server: a new job arrived.....");

        exe.execute(task);
        System.out.println("server: pool size:"+exe.getPoolSize());//no of thread in pool

        System.out.println("server: Active count:"+exe.getActiveCount());//no of active threads

        System.out.println("server: Completed task:"+exe.getCompletedTaskCount());//no of task completed

    }

    public void endServer()
    {
        exe.shutdown();
    }
}
```

```
public class ExecutorDemoWithRunnable {

    public static void main(String[] args) {
        Server server=new Server();

        for(int i=0;i<10;i++)
        {
            Task t=new Task("task "+i);
            server.executeTask(t);//providing job to executor framework
        }

        server.endServer();
    }
}
```

# Executor with <<Callable>>

- One advantage of Executor framework that We can use <<Callable>> override call() method that can return results
- <<Future>> helps us to retrieve results

```
class FactorialCalculator implements Callable<Integer>
{
    private Integer number;
    FactorialCalculator(Integer n){number=n;}

    @Override
    public Integer call() throws Exception
    {
        int result=1;
        if(number==0 || number==1)
            result=1;
        else
        {
            for(int i=2;i<=number;i++)
            {
                result=result*i;

                Thread.sleep(100); //assume heavy processing
            }
        }

        System.out.println("result:"+result);
        return result;
    }
}
```

```
public static void main(String[] args) throws InterruptedException, ExecutionException {
    ThreadPoolExecutor exe=(ThreadPoolExecutor) Executors.newFixedThreadPool(2);
    for(int i=3;i<30;i++)
    {
        FactorialCalculator cal=new FactorialCalculator(i);
        Future<Integer>result=exe.submit(cal);
        System.out.println(result.get());
    }
    System.out.println("finished all jobs.....");
    exe.shutdown();
}
```

```
public static void main(String[] args) {  
    Thread thread1 = new Thread(new Task());  
    thread1.start();  
    System.out.println("Thread Name: " + Thread.currentThread().getName());  
}  
  
static class Task implements Runnable {  
    public void run() {  
        System.out.println("Thread Name: " + Thread.currentThread().getName());  
    }  
}
```

Running a task asynchronously

main thread

t1.start()

sout (in main)

thread-0

sout (in other thread)



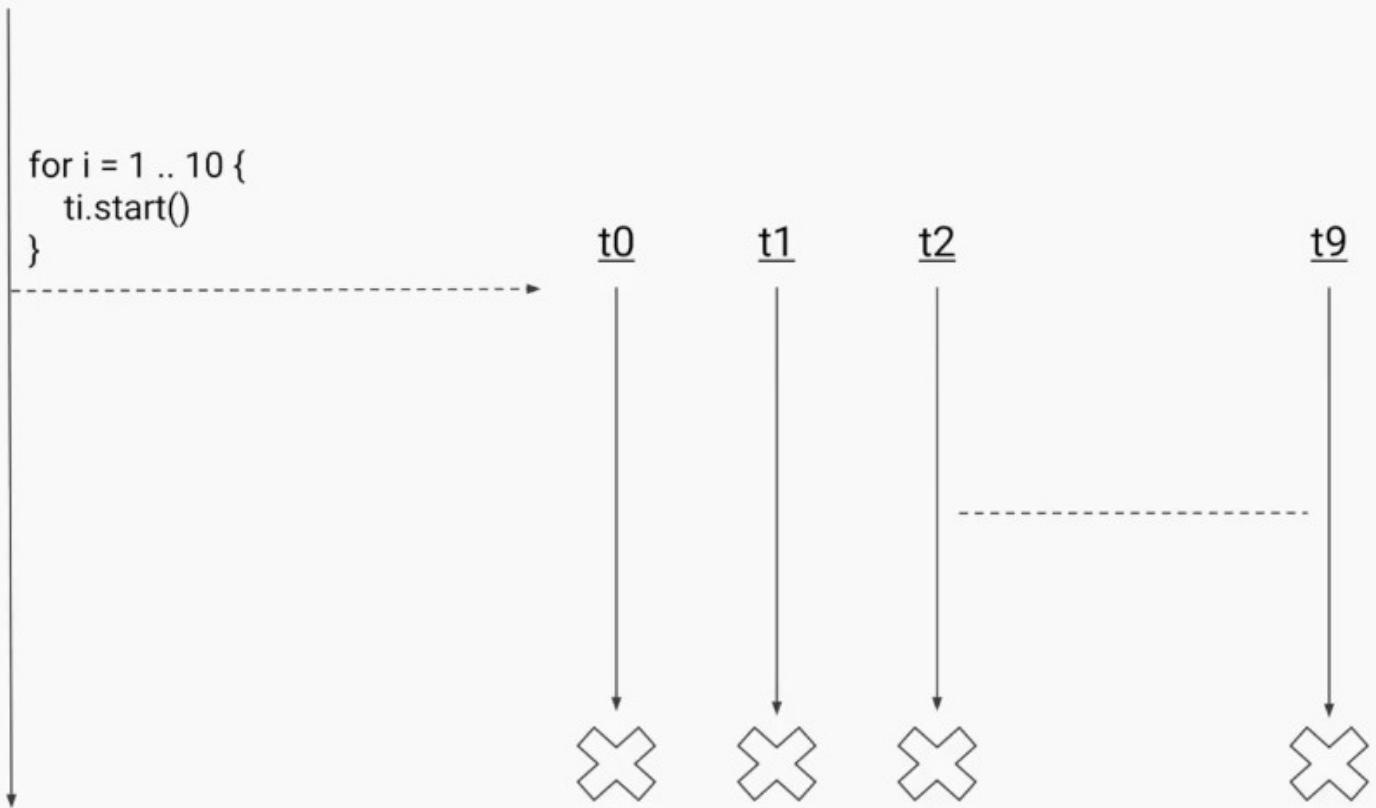
Running multiple tasks asynchronously

```
public static void main(String[] args) {  
  
    for (int i = 0; i < 10; i++) {  
        Thread thread = new Thread(new Task());  
        thread.start();  
    }  
    System.out.println("Thread Name: " + Thread.currentThread().getName());  
}  
  
static class Task implements Runnable {  
    public void run() {  
        System.out.println("Thread Name: " + Thread.currentThread().getName())  
    }  
}
```

Running a task asynchronously

## main thread

```
for i = 1 .. 10 {  
    ti.start()  
}
```

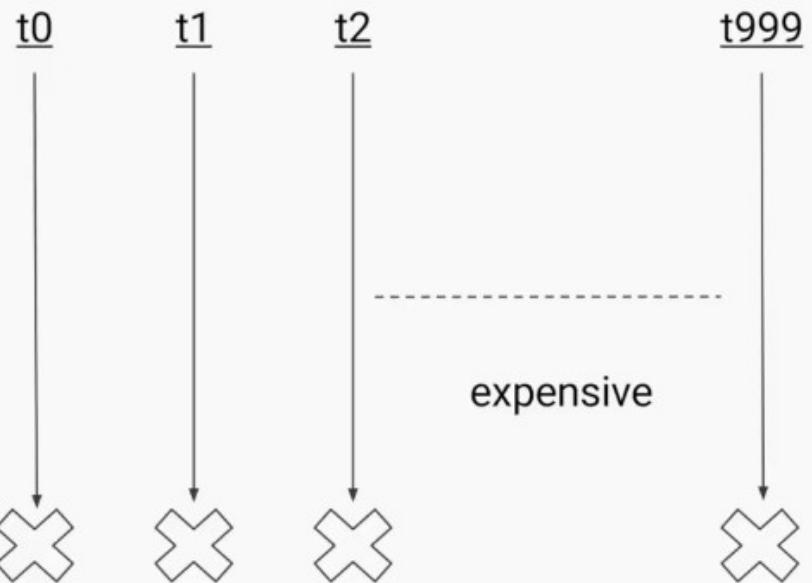


Running multiple tasks asynchronously

### main thread

```
for i = 1 .. 1000 {  
    ti.start()  
}
```

1 Java thread = 1 OS thread



Running 1000s of tasks asynchronously

## main thread

create pool

```
for i = 1 .. 1000 {  
    submit-tasks  
}
```

Fixed number of threads

t0

t1

t2

t9

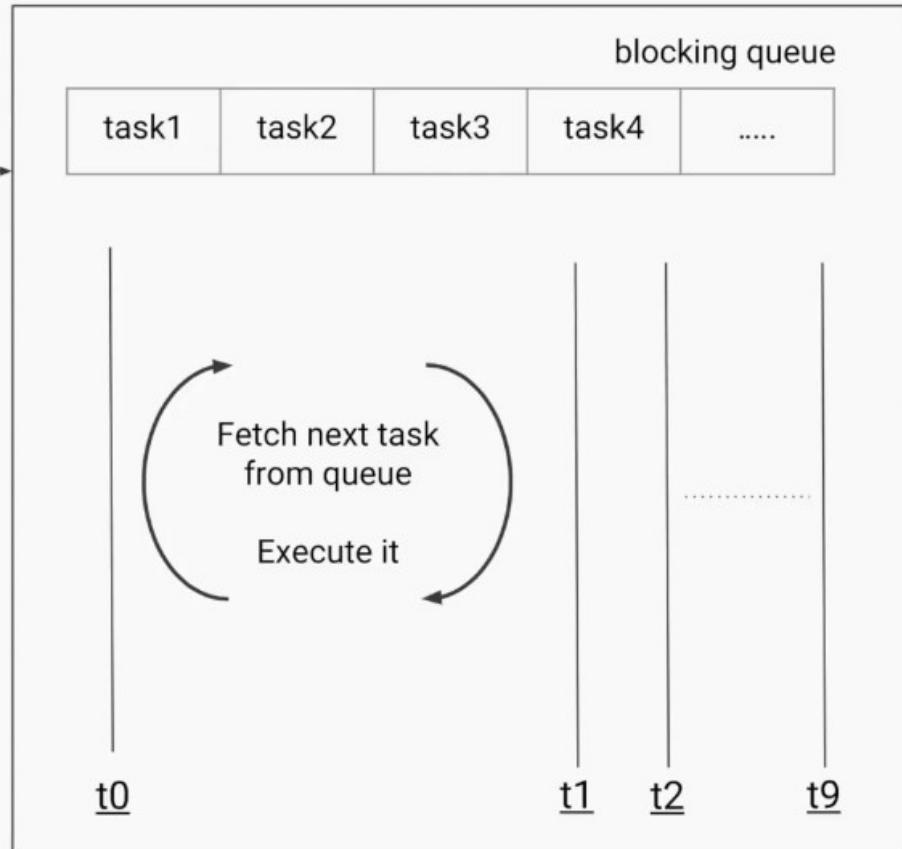
```
public static void main(String[] args) {  
  
    // create the pool  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new Task());  
    }  
    System.out.println("Thread Name: " + Thread.currentThread().getName());  
}  
  
static class Task implements Runnable {  
    public void run() {  
        System.out.println("Thread Name: " + Thread.currentThread().getName());  
    }  
}
```

Submit tasks using ThreadPool

### main thread

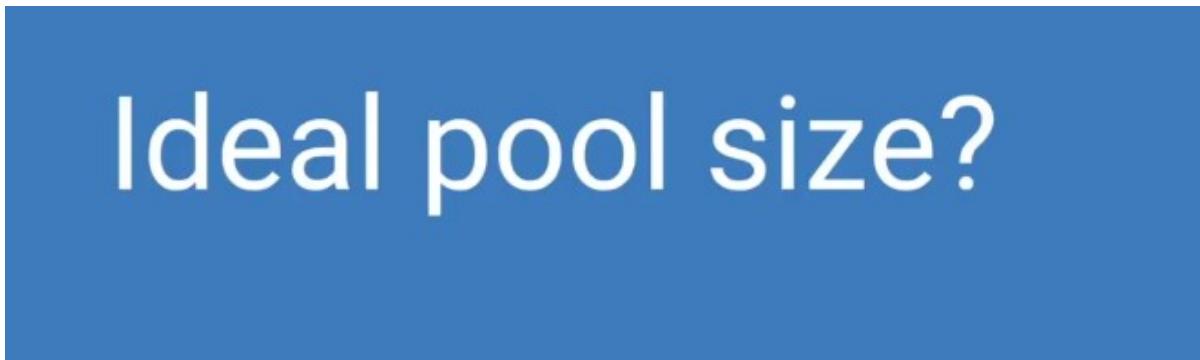
```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```

### thread-pool



How thread pool internally works

- 
1. FixedThreadPool
  2. CachedThreadPool
  3. ScheduledThreadPool
  4. SingleThreadedExecutor

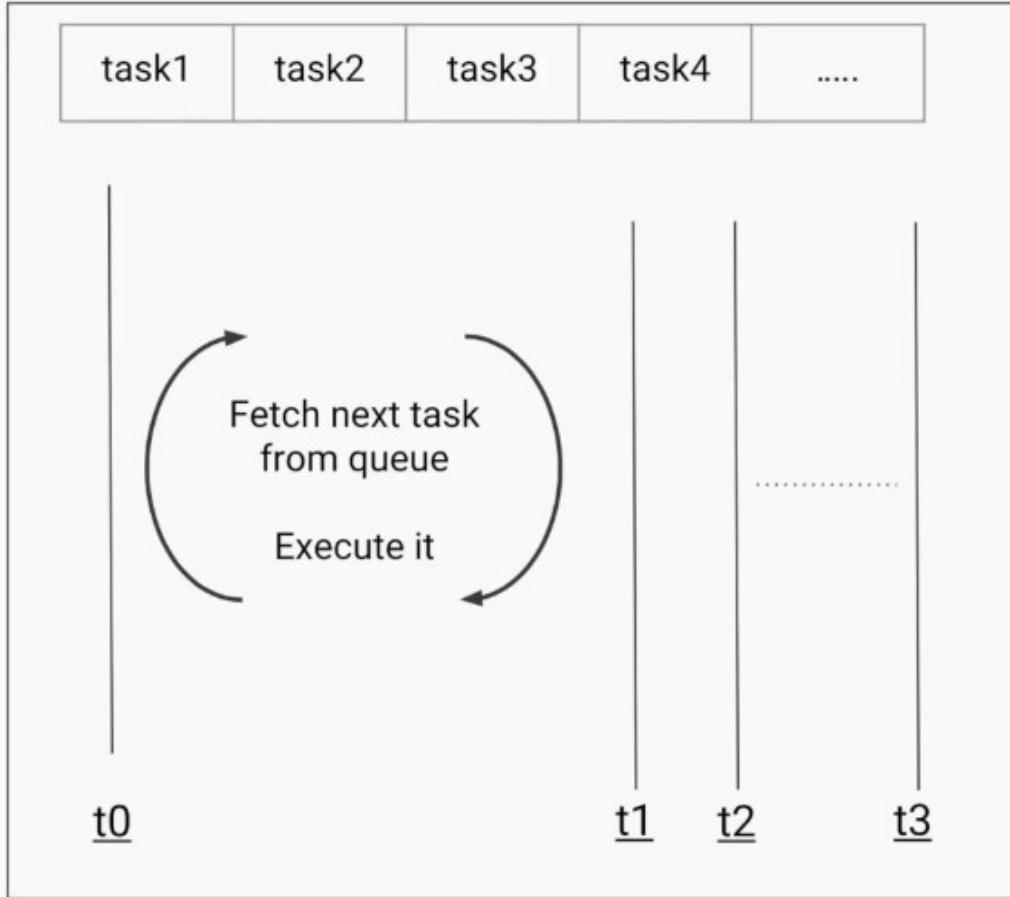


Ideal pool size?

| Task Type     | Ideal pool size | Considerations                                                                                                                                 |
|---------------|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| CPU intensive | CPU Core count  | How many other applications (or other executors/threads) are running on the same CPU.                                                          |
| IO intensive  | High            | Exact number will depend on rate of task submissions and average task wait time.<br><br>Too many threads will increase memory consumption too. |

Pool size depends on the task type

## thread-pool



CPU

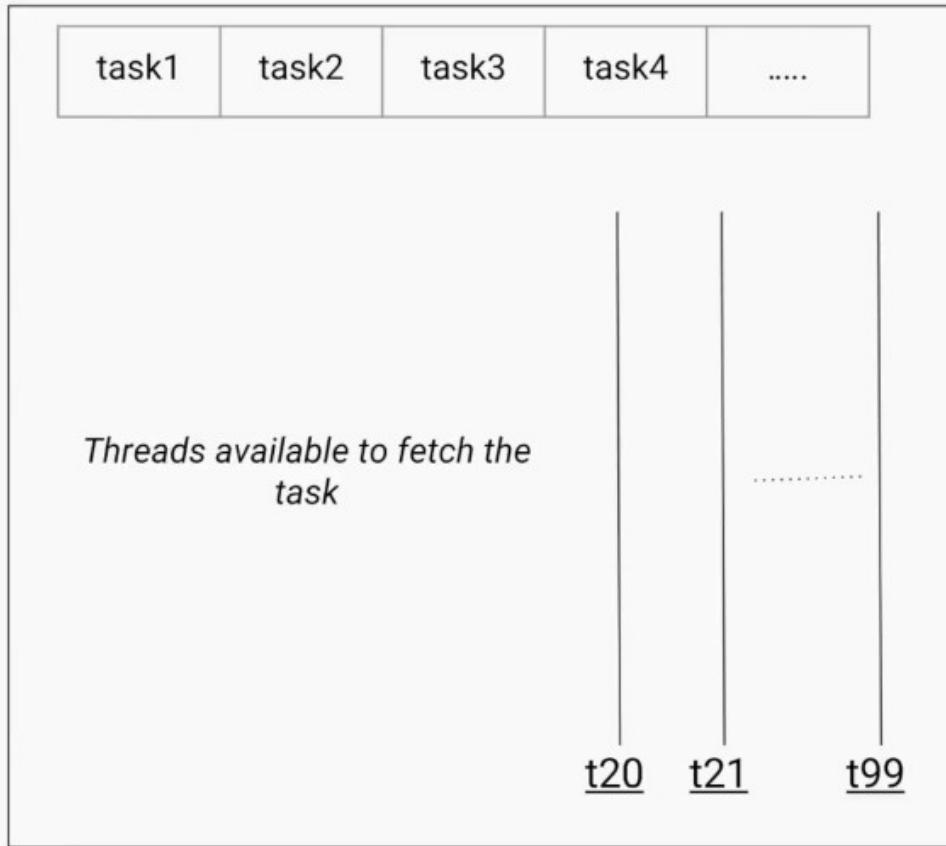
|        |        |
|--------|--------|
| Core 1 | Core 2 |
| Core 3 | Core 4 |

*Max 4 threads can run at a time*

```
public static void main(String[] args) {  
  
    // get count of available cores  
    int coreCount = Runtime.getRuntime().availableProcessors();  
    ExecutorService service = Executors.newFixedThreadPool(coreCount);  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new CpuIntensiveTask());  
    }  
}  
  
static class CpuIntensiveTask implements Runnable {  
    public void run() {  
        // some CPU intensive operations  
    }  
}
```

Pool size for CPU intensive Tasks

## thread-pool



## CPU

|        |        |
|--------|--------|
| Core 1 | Core 2 |
| Core 3 | Core 4 |

*Max 4 threads can run at a time*

## Waiting threads

|    |    |    |    |      |
|----|----|----|----|------|
| t3 | t5 | t6 | t7 | .... |
|----|----|----|----|------|

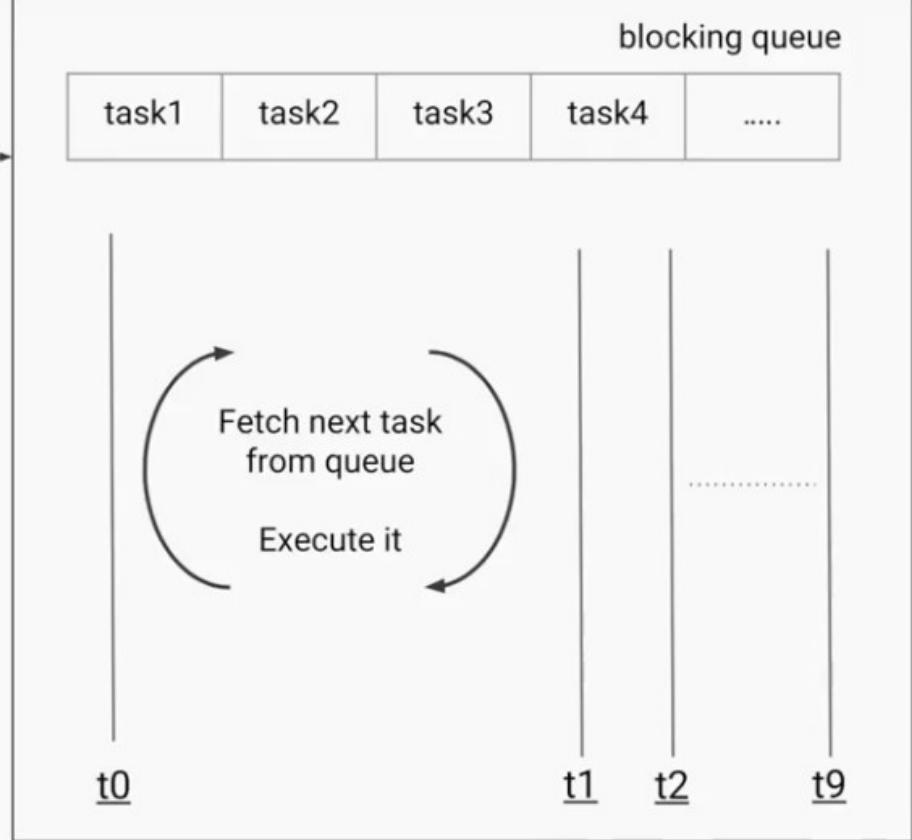
*Threads waiting for IO operation response from the OS*

```
public static void main(String[] args) {  
    // much higher count for IO tasks  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 100 );  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new IOTask());  
    }  
}  
  
static class IOTask implements Runnable {  
    public void run() {  
        // some IO operations which will cause thread to block/wait  
    }  
}
```

### main thread

```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```

### thread-pool



How thread pool internally works

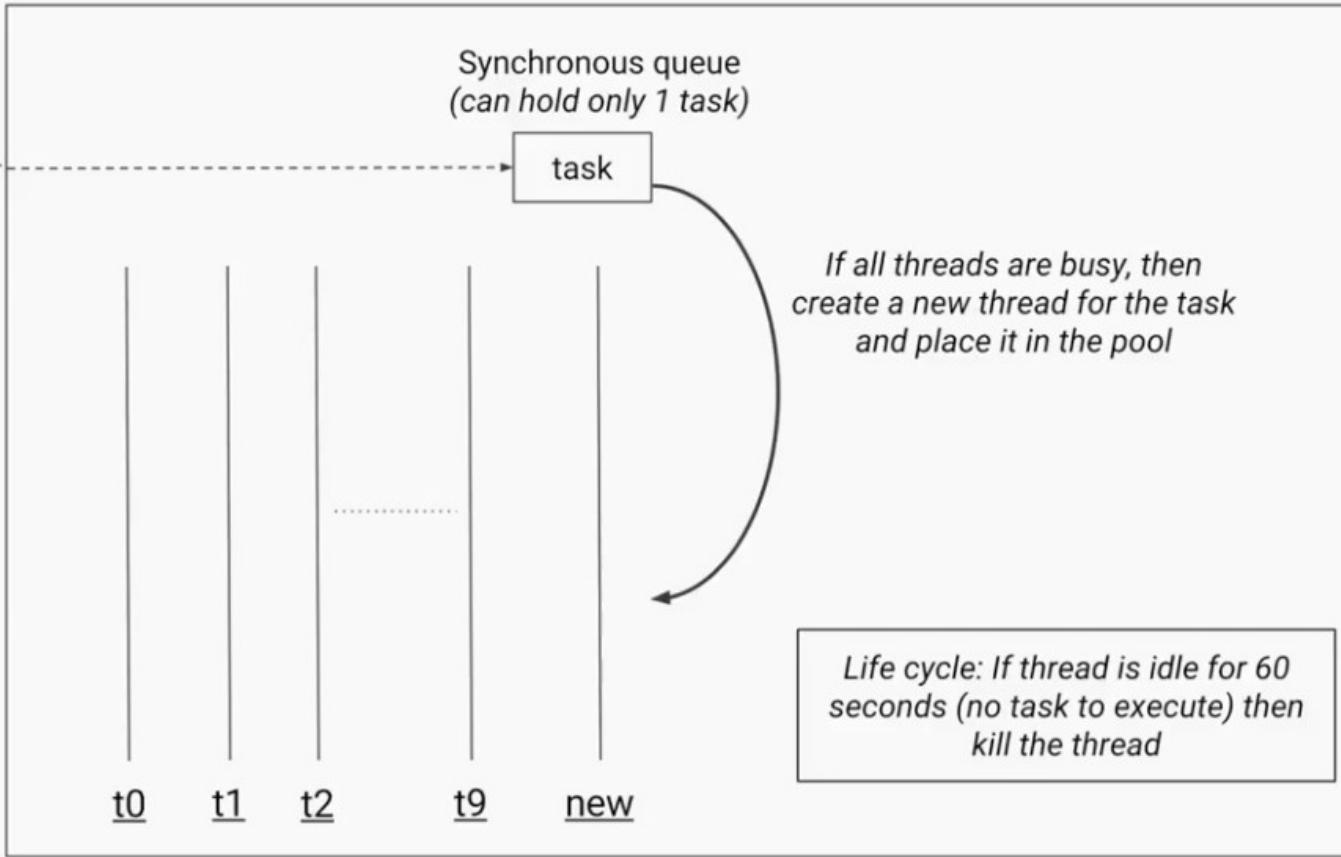
```
public static void main(String[] args) {  
    // create the pool  
    ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new Task());  
    }  
    System.out.println("Thread Name: " + Thread.currentThread().getName());  
}  
  
static class Task implements Runnable {  
    public void run() {  
        System.out.println("Thread Name: " + Thread.currentThread().getName())  
    }  
}
```

Submit tasks using ThreadPool

## main-thread

```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```

## thread-pool



```
public static void main(String[] args) {  
  
    // for lot of short lived tasks  
    ExecutorService service = Executors.newCachedThreadPool();  
  
    // submit the tasks for execution  
    for (int i = 0; i < 100; i++) {  
        service.execute(new Task());  
    }  
}  
  
static class Task implements Runnable {  
    public void run() {  
        // short lived task  
    }  
}
```

## main-thread

Service.schedule

service.scheduleAtFixedRate

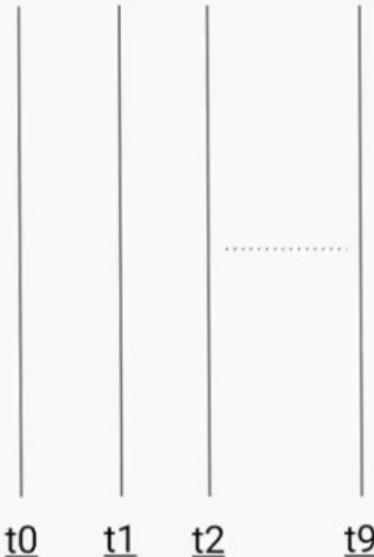
service.scheduleAtFixedDelay

## thread-pool

Delay queue

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| task1 | task2 | task3 | task4 | ..... |
|-------|-------|-------|-------|-------|

*Schedule the tasks to run based on time delay (and retrigger for fixedRate / fixedDelay)*



*Life Cycle: More threads are created if required.*

Scheduled Thread Pool =



- Pool sizes changes
- Queue types
- Task rejections
- Life cycle methods

# ThreadPoolExecutor Constructor:customization

```
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );
```



```
public static ExecutorService newFixedThreadPool(int nThreads) {  
    return new ThreadPoolExecutor(nThreads, nThreads,  
        keepAliveTime: 0L, TimeUnit.MILLISECONDS,  
        new LinkedBlockingQueue<Runnable>());  
}
```

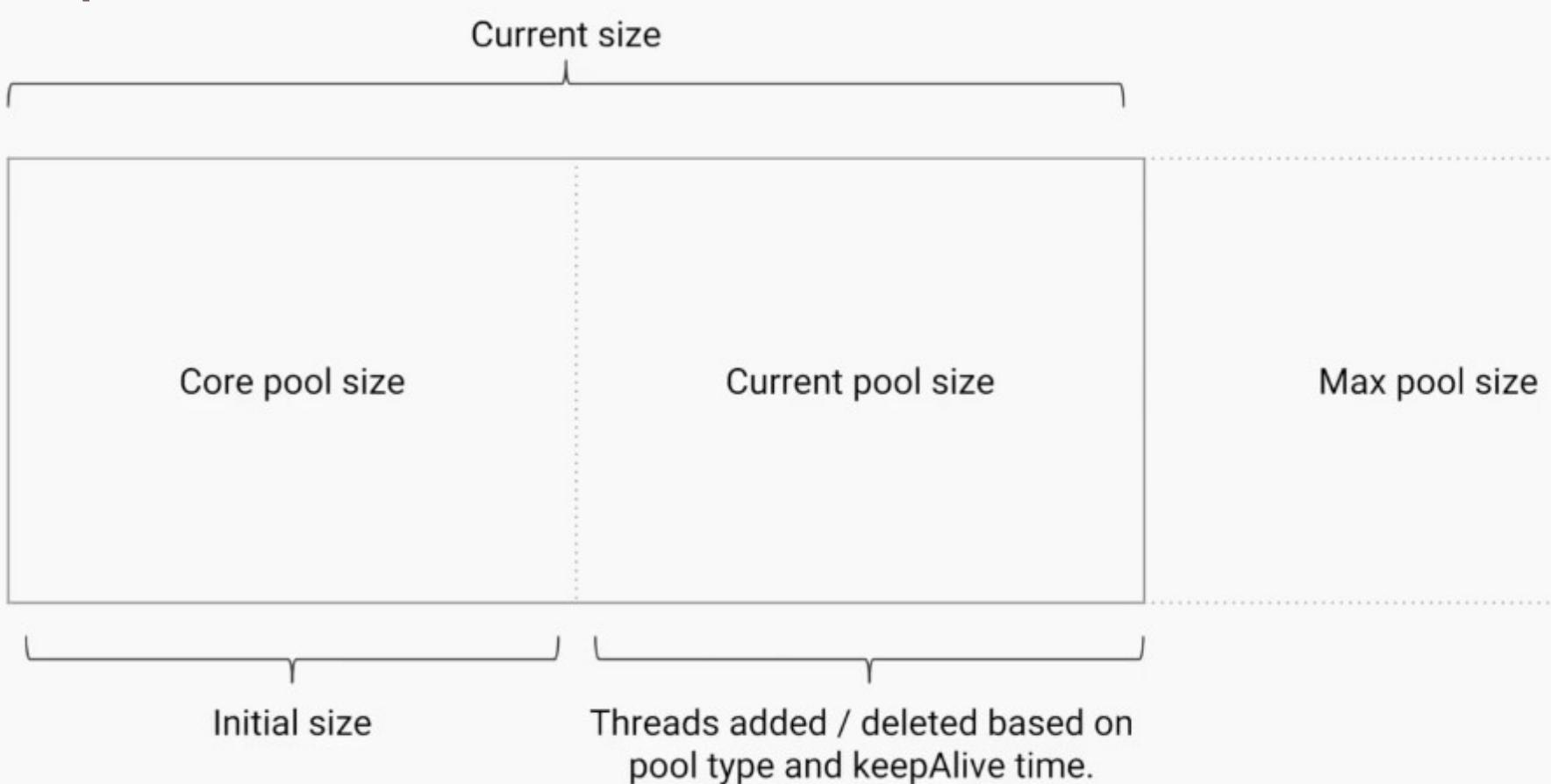


```
public ThreadPoolExecutor(int corePoolSize,  
    int maximumPoolSize,  
    long keepAliveTime,  
    TimeUnit unit,  
    BlockingQueue<Runnable> workQueue,  
    ThreadFactory threadFactory,  
    RejectedExecutionHandler handler) {
```

# ThreadPoolExecutor Constructor parameters

| Parameter            | Type                     | Meaning                                                      |
|----------------------|--------------------------|--------------------------------------------------------------|
| corePoolSize         | int                      | Minimum/Base size of the pool                                |
| maxPoolSize          | int                      | Maximum size of the pool                                     |
| keepAliveTime + unit | long                     | Time to keep an idle thread alive (after which it is killed) |
| workQueue            | BlockingQueue            | Queue to store the tasks from which threads fetch them       |
| threadFactory        | ThreadFactory            | The factory to use to create new threads                     |
| handler              | RejectedExecutionHandler | Callback to use when tasks submitted are rejected            |

# ThreadPoolExecutor :understanding pool size



# ThreadPoolExecutor :understanding pool size

| Parameter     | FixedThreadPool      | CachedThreadPool  | ScheduledThreadPool | SingleThreaded |
|---------------|----------------------|-------------------|---------------------|----------------|
| corePoolSize  | constructor-arg      | 0                 | constructor-arg     | 1              |
| maxPoolSize   | same as corePoolSize | Integer.MAX_VALUE | Integer.MAX_VALUE   | 1              |
| keepAliveTime | 0 seconds            | 60 seconds        | 60 seconds          | 0 seconds      |

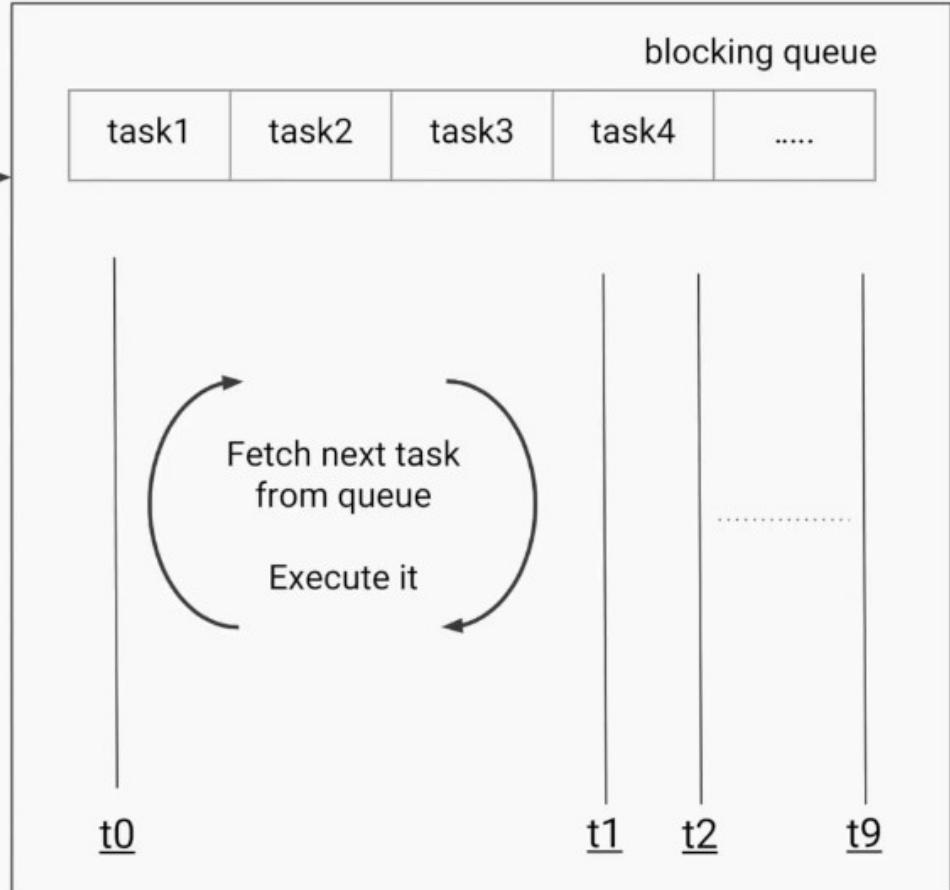
*Note: Core pool threads are never killed unless  
allowCoreThreadTimeOut(boolean value) is set to true.*

# Type of Queues used in thread pool

main thread

```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```

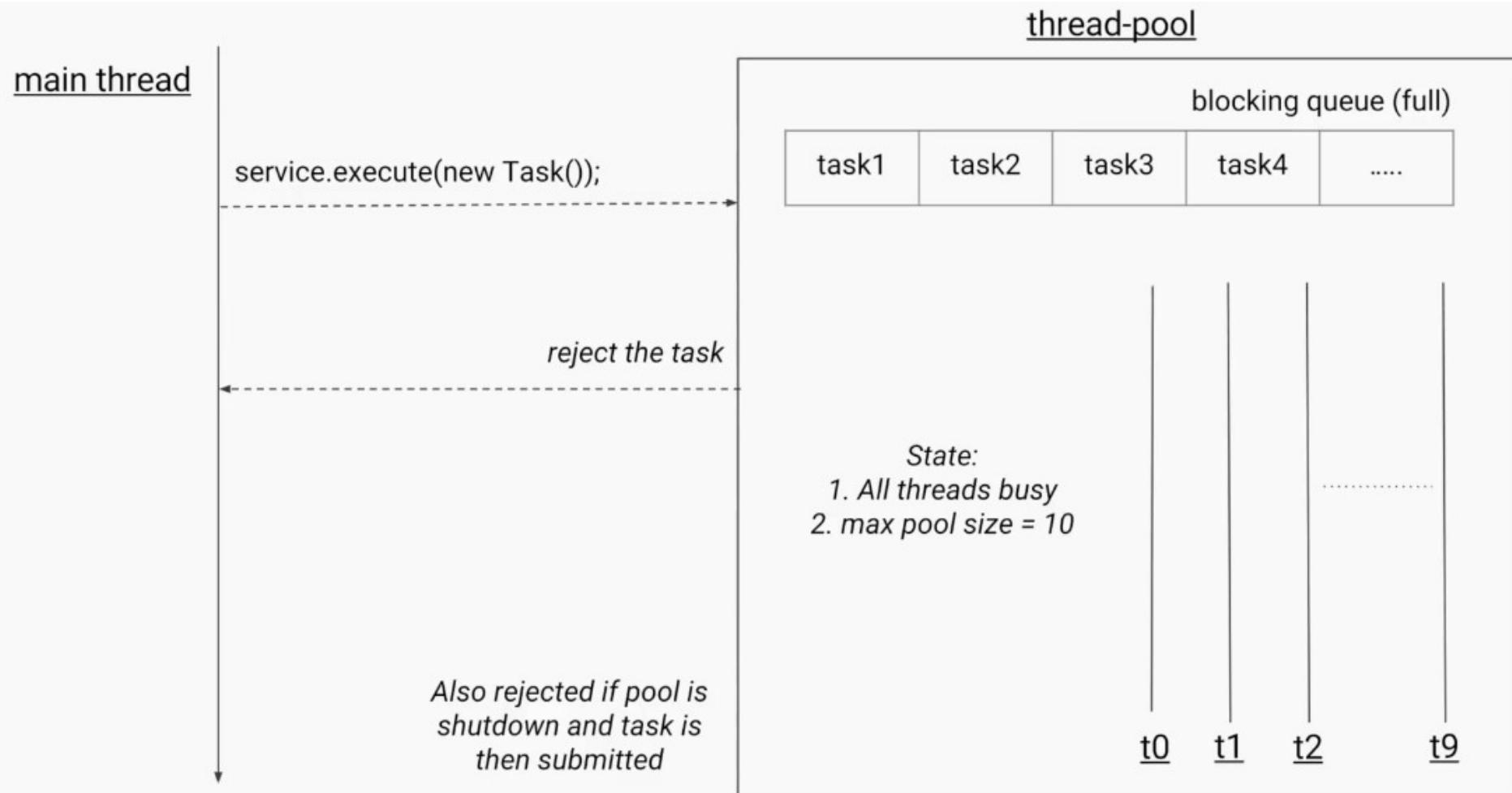
thread-pool



# ThreadPoolExecutor :understanding Queue types

| Pool                 | Queue Type          | Why?                                                                                                                     |
|----------------------|---------------------|--------------------------------------------------------------------------------------------------------------------------|
| FixedThreadPool      | LinkedBlockingQueue | Threads are limited, thus unbounded queue to store all tasks.                                                            |
| SingleThreadExecutor | LinkedBlockingQueue | <i>Note: Since queue can never become full, new threads are never created.</i>                                           |
| CachedThreadPool     | SynchronousQueue    | Threads are unbounded, thus no need to store the tasks.<br>Synchronous queue is a queue with single slot                 |
| ScheduledThreadPool  | DelayedWorkQueue    | Special queue that deals with schedules/time-delays                                                                      |
|                      |                     |                                                                                                                          |
| Custom               | ArrayBlockingQueue  | Bounded queue to store the tasks. If queue gets full, new thread is created (as long as count is less than maxPoolSize). |

# thread pool: handling rejections



# thread pool: rejection policies

| Policy              | What it means?                                                                                                                                                                                 |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AbortPolicy         | Submitting new tasks throws RejectedExecutionException (Runtime exception)                                                                                                                     |
| DiscardPolicy       | Submitting new tasks silently discards it.                                                                                                                                                     |
| DiscardOldestPolicy | Submitting new tasks drops existing oldest task, and new task is added to the queue.                                                                                                           |
| CallerRunsPolicy    | Submitting new tasks will execute the task on the caller thread itself. This can create feedback loop where caller thread is busy executing the task and cannot submit new tasks at fast pace. |

# thread pool: RejectedExecutionException

```
ExecutorService service
    = new ThreadPoolExecutor(
        corePoolSize: 10,
        maximumPoolSize: 100,
        keepAliveTime: 120, TimeUnit.SECONDS,
        new ArrayBlockingQueue<>( capacity: 300));

try {
    service.execute(new Task());
} catch (RejectedExecutionException e) {
    System.err.println("task rejected " + e.getMessage());
}
```

# thread pool: RejectedExecutionException

```
ExecutorService service
    = new ThreadPoolExecutor(
        corePoolSize: 10,
        maximumPoolSize: 100,
        keepAliveTime: 120, TimeUnit.SECONDS,
        new ArrayBlockingQueue<>( capacity: 300),
        new CustomRejectionHandler());
    •

private static class CustomRejectionHandler implements RejectedExecutionHandler{
    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        // logging / operations to perform on rejection
    }
}
```

# thread pool: Life Cycle method

```
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );
for (int i = 0; i < 100; i++) {
    service.execute(new Task());
}

// initiate shutdown
service.shutdown();

// will throw RejectionExecutionException
// service.execute(new Task());

// will return true, since shutdown has begun
service.isShutdown();

// will return true if all tasks are completed
// including queued ones
service.isTerminated();

// block until all tasks are completed or if timeout occurs
service.awaitTermination( timeout: 10, TimeUnit.SECONDS );

// will initiate shutdown and return all queued tasks
List<Runnable> runnables = service.shutdownNow();
```

# thread pool: Example

```
// create the pool
ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );

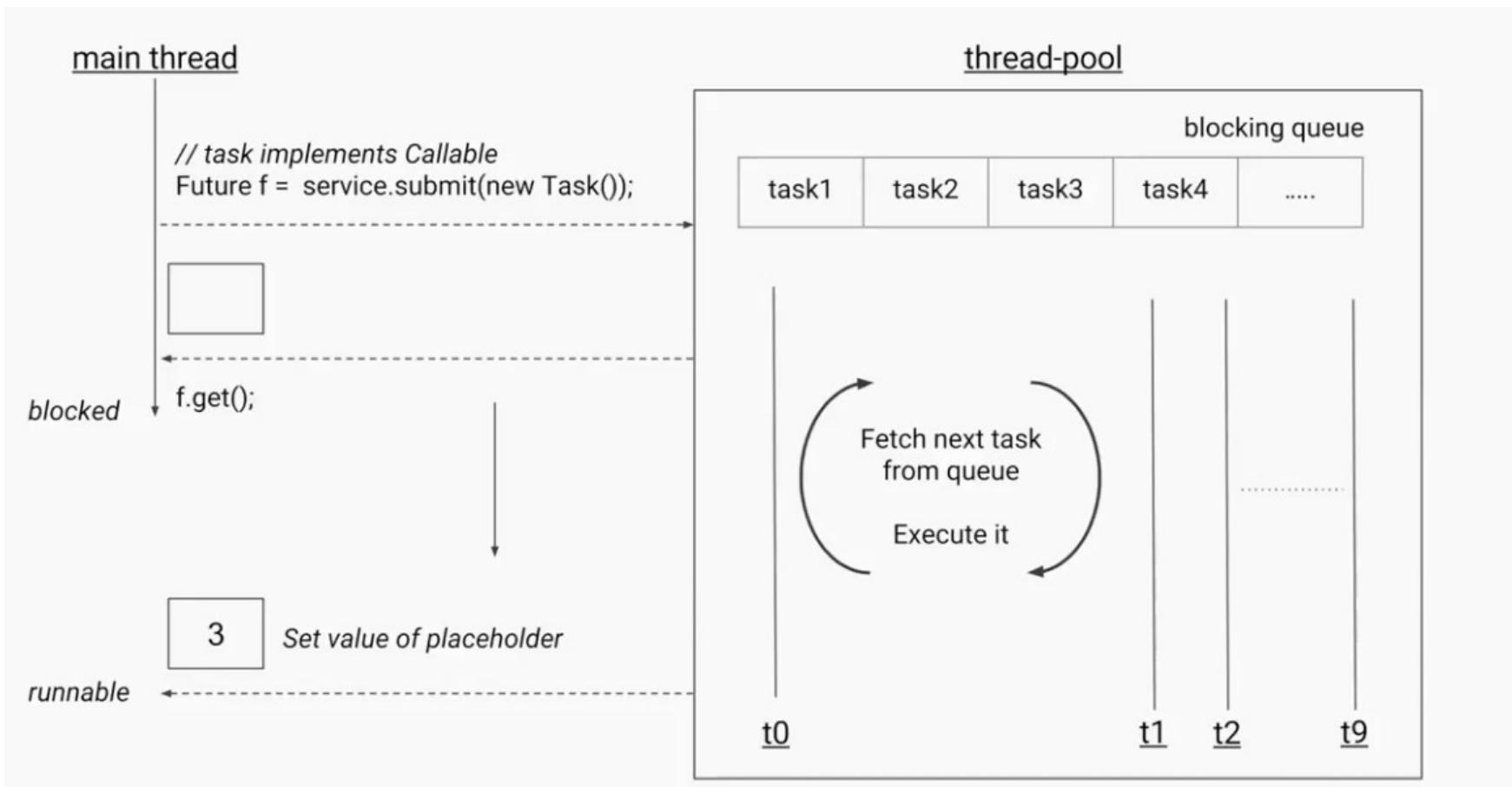
// submit the tasks for execution
List<Future> allFutures = new ArrayList<>();
for (int i = 0; i < 100; i++) {
    Future<Integer> future = service.submit(new Task());
    allFutures.add(future);
}

// 100 futures, with 100 placeholders.

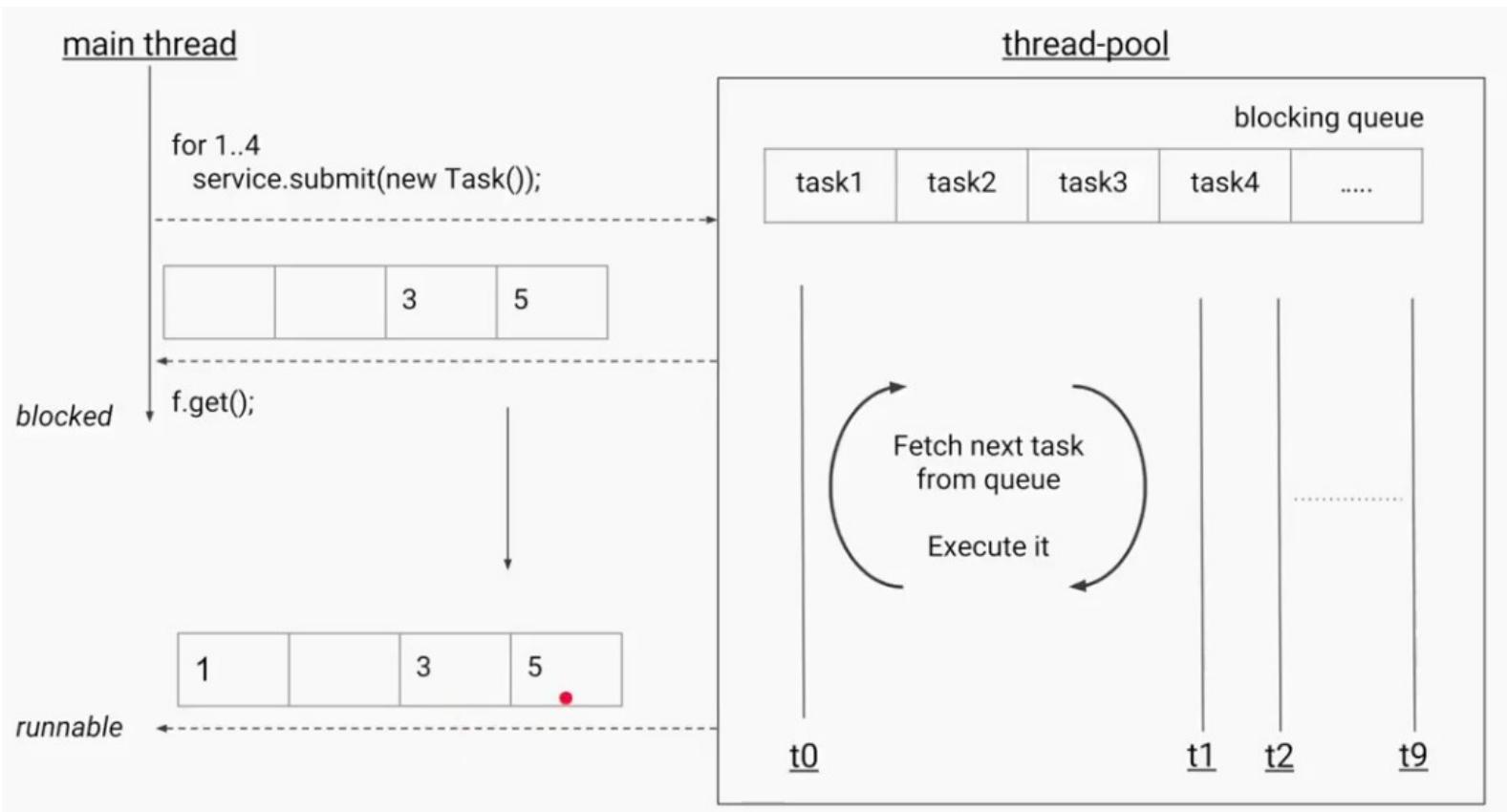
// perform some unrelated operations

// 1 sec
try {
    Integer result = future.get(); // blocking
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
}
```

# thread pool: Future how it work?



# thread pool: Blocked till all work is not done! Caution



```
public static void main(String[] args) {  
  
    // for scheduling of tasks  
    ScheduledExecutorService service = Executors.newScheduledThreadPool( corePoolSize: 10 );  
  
    // task to run after 10 second delay  
    service.schedule(new Task(), delay: 10, SECONDS);  
  
    // task to run repeatedly every 10 seconds  
    service.scheduleAtFixedRate(new Task(), initialDelay: 15, period: 10, SECONDS);  
  
    // task to run repeatedly 10 seconds after previous task completes  
    service.scheduleWithFixedDelay(new Task(), initialDelay: 15, delay: 10, TimeUnit.SECONDS);  
  
}  
  
static class Task implements Runnable {  
    public void run() {  
        // task that needs to run  
        // based on schedule  
    }  
}
```

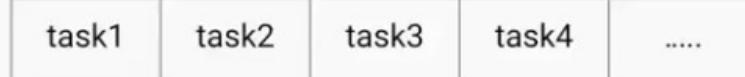
Scheduled Executor

### main thread

```
for i = 1 .. 100 {  
    service.execute(new Task());  
}
```

### thread-pool

#### blocking queue



Fetch next task  
from queue

Execute it

t0

*Life Cycle: Recreates thread if killed  
because of the task.*



**CompletableFuture**

# Need for speed

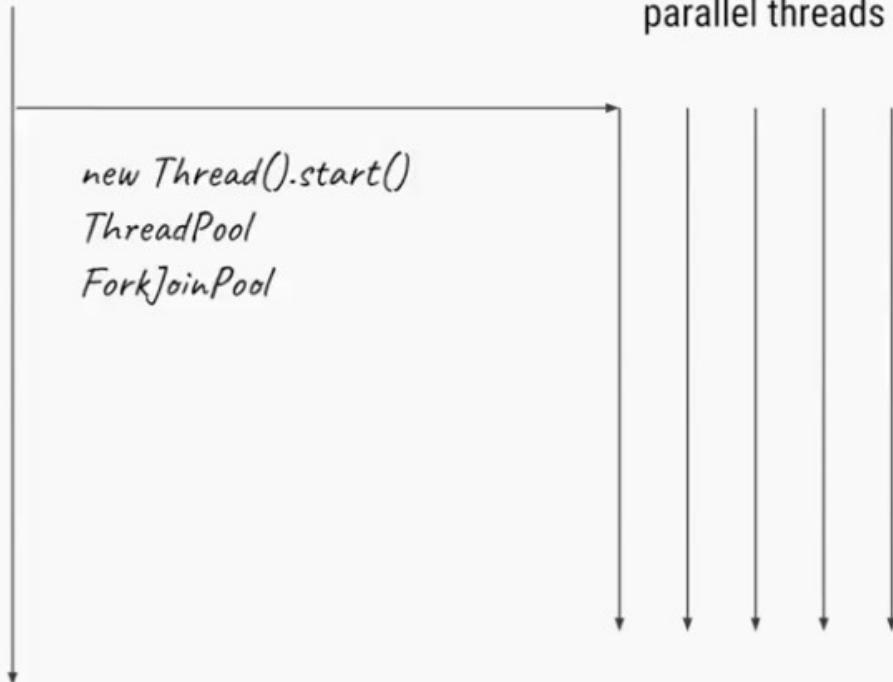
main thread

parallel threads

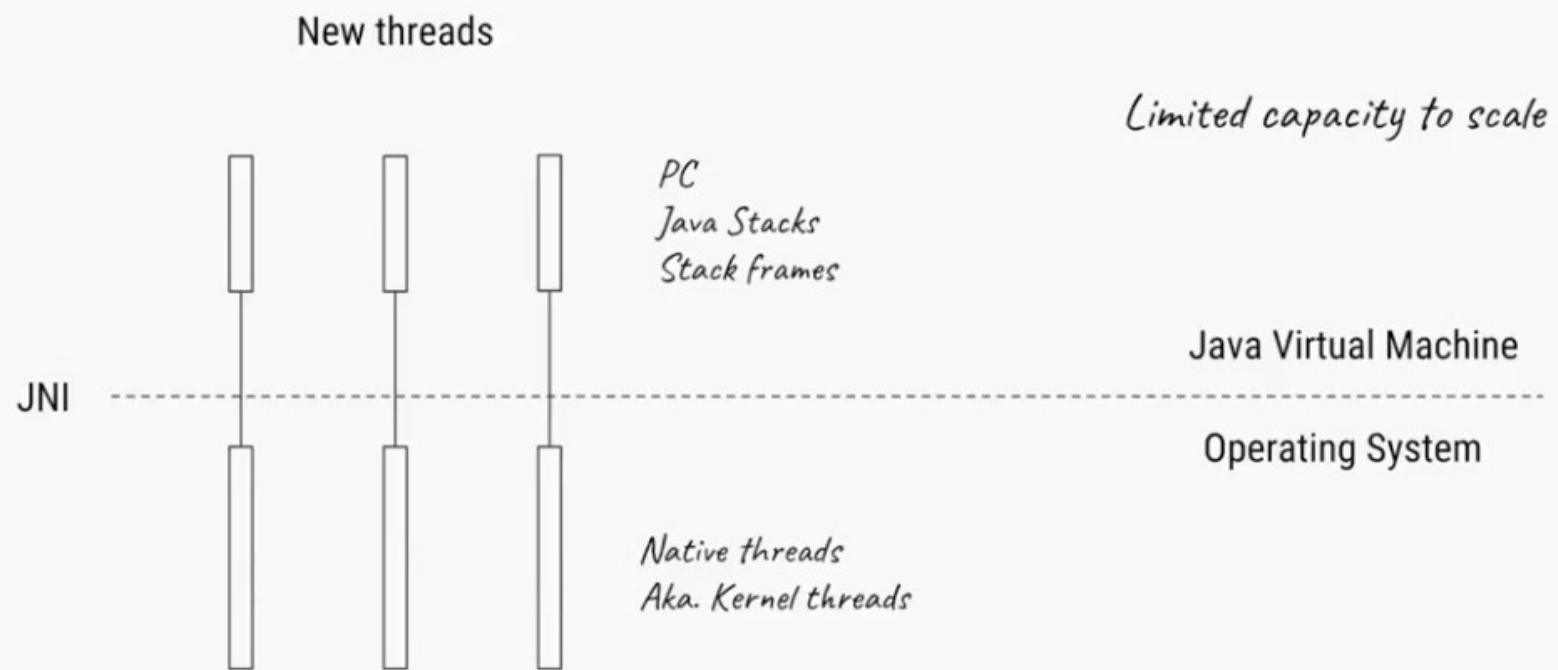
```
new Thread().start()  
ThreadPool  
ForkJoinPool
```

|        |        |
|--------|--------|
| Core 1 | Core 2 |
| Core 3 | Core 4 |

CPU



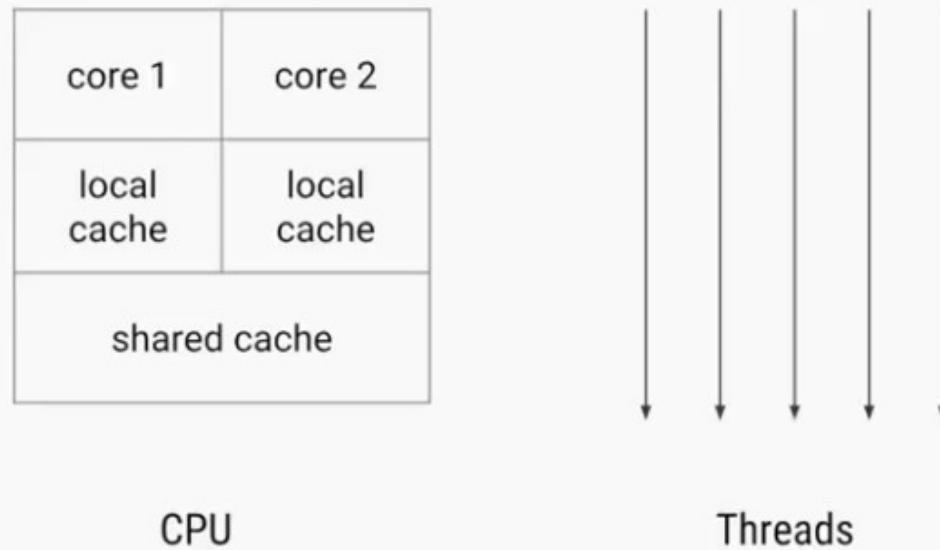
## Java thread = OS thread



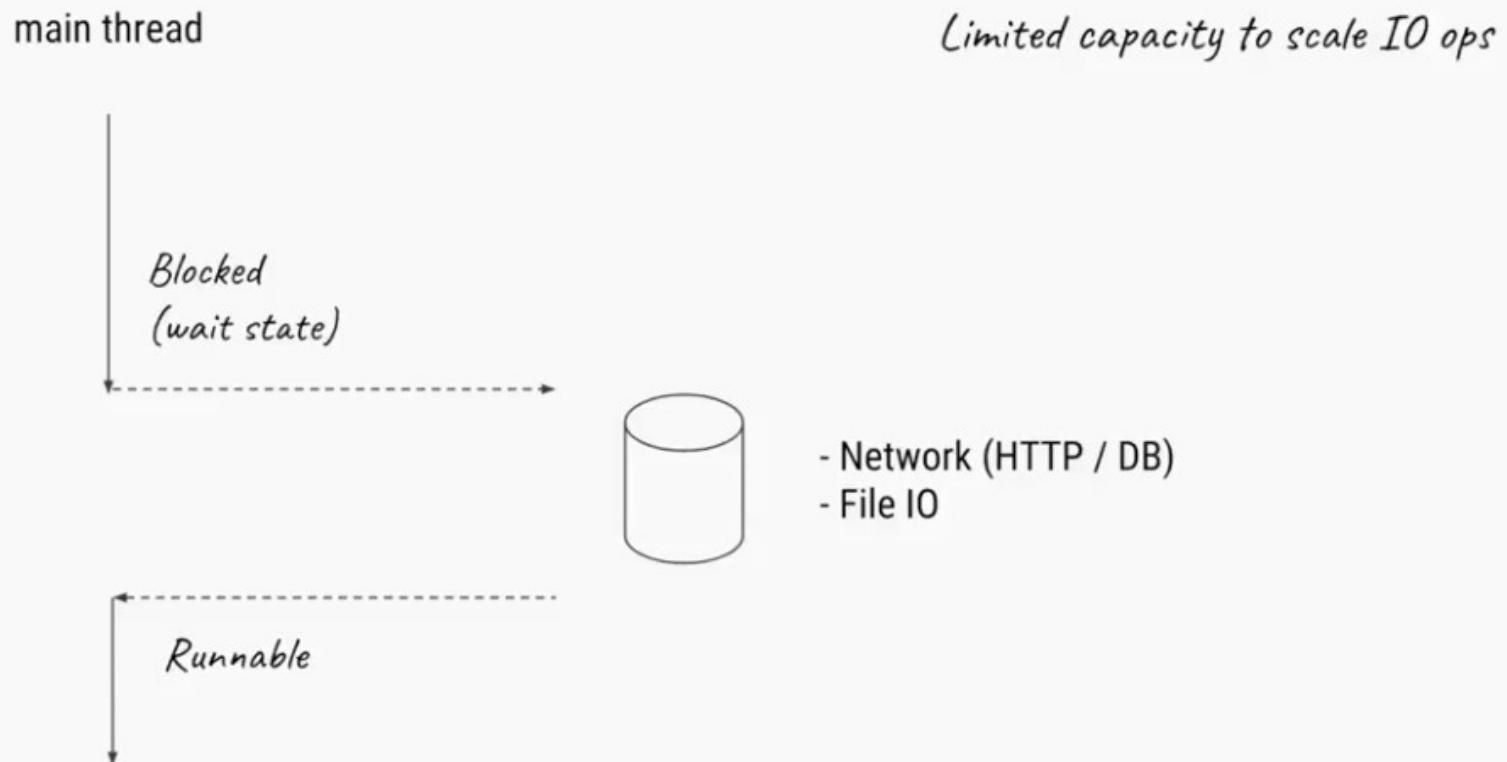
## Other issues too...

*Scheduling Overhead,*

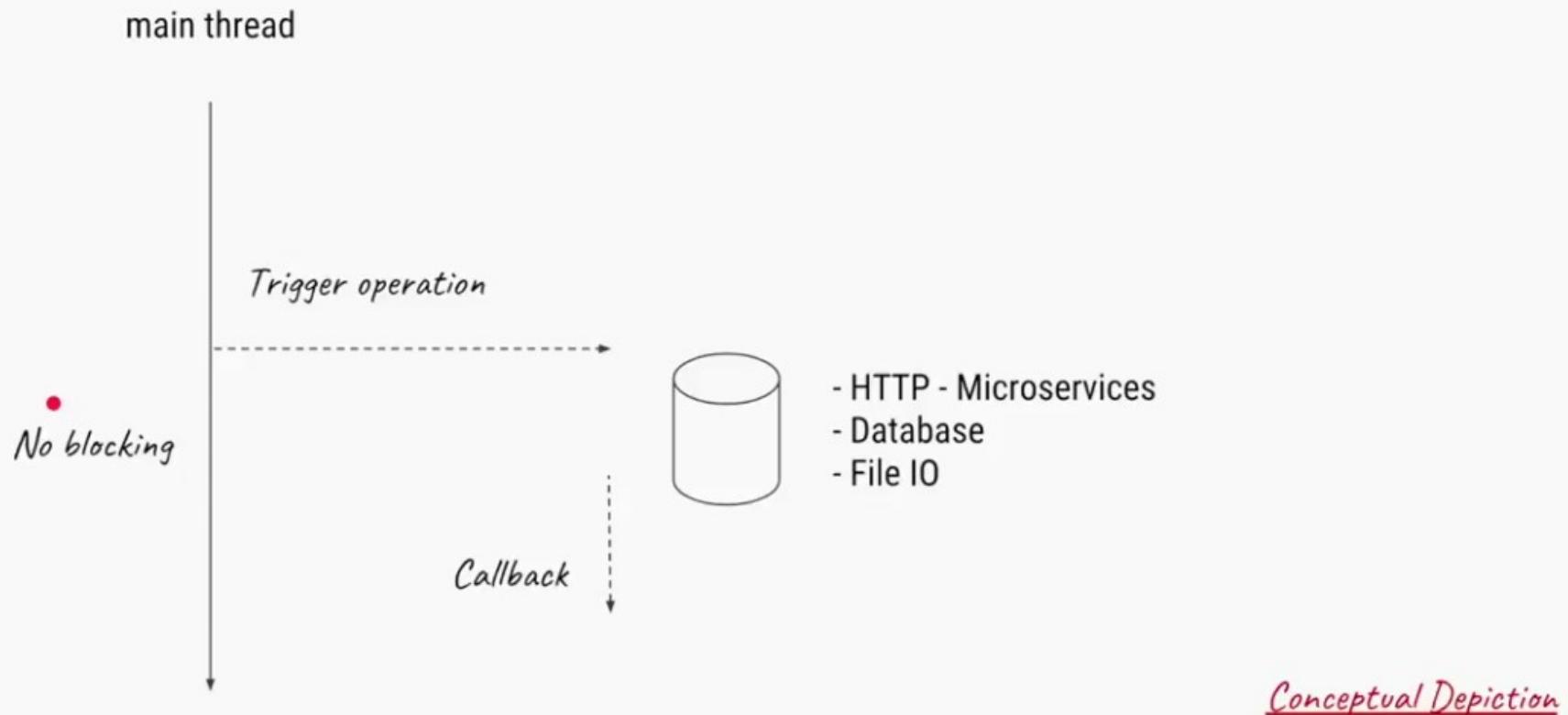
*Data Locality*



## IO operations require more threads



# Non-blocking IO



## Synchronous API

```
for (Integer id : employeeIds) {  
  
    // Step 1: Fetch Employee details from DB  
    Future<Employee> future = service.submit(new EmployeeFetcher(id));  
    Employee emp = future.get(); // blocking  
  
    // Step 2: Fetch Employee tax rate from REST service  
    Future<TaxRate> rateFuture = service.submit(new TaxRateFetcher(emp));  
    TaxRate taxRate = rateFuture.get(); // blocking  
  
    // Step 3: Calculate current year tax  
    BigDecimal tax = calculateTax(emp, taxRate);  
  
    // Step 4: Send email to employee using REST service  
    service.submit(new SendEmail(emp, tax));  
}
```

## Asynchronous API - Callbacks

```
for (Integer id : employeeIds) {  
    CompletableFuture.supplyAsync(() -> fetchEmployee(id))  
        .thenApplyAsync(employee -> fetchTaxRate(employee))  
        .thenApplyAsync(taxRate -> calculateTax(taxRate))  
        .thenAcceptAsync(taxValue -> sendEmail(taxValue));  
}
```

*Callback chaining (similar to JS)*

## Java NIO

- Buffers, Channels, Selectors
- Low-level API for asynchronous / non-blocking IO
- Applicable for Files, Sockets,
- Listener based (callbacks)

```
ByteBuffer buffer = ByteBuffer.allocate(1024);

Path path = Paths.get("/home/file2");
AsynchronousFileChannel fileChannel = AsynchronousFileChannel.open(path, StandardOpenOption.READ);

fileChannel.read(buffer, position: 0, buffer, new CompletionHandler<Integer, ByteBuffer>() {

    @Override
    public void completed(Integer result, ByteBuffer data) {           <= Callback
        // process data|
    }
})
```

## Spring - Reactive Programming

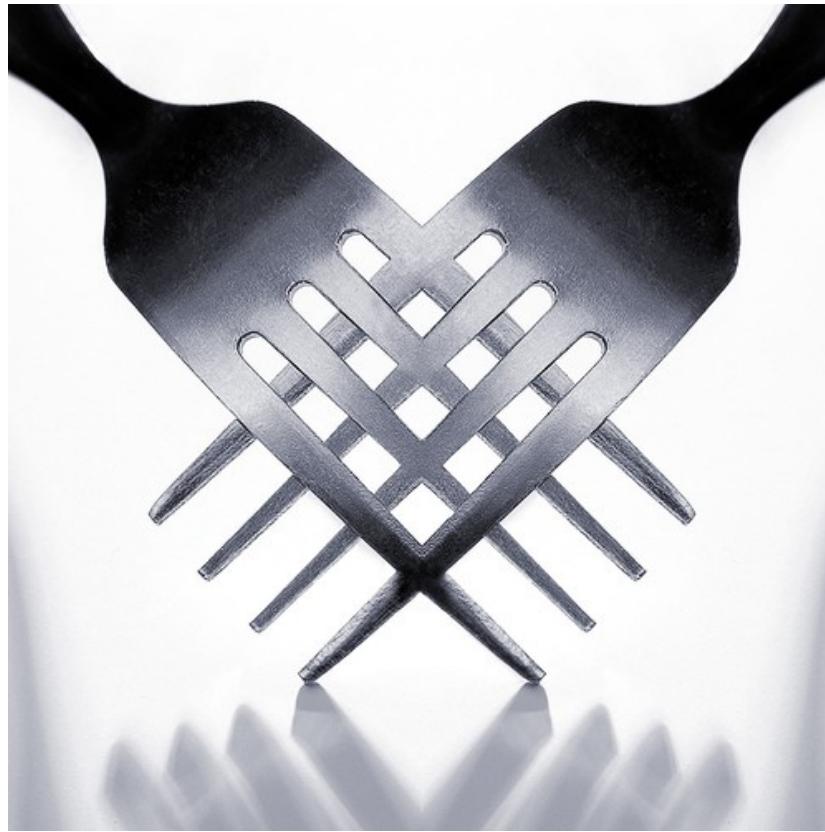
```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public Mono<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

# Fork-Join framework

---



# Fork Join Framework

## Parallel version of Divide and Conquer

1. Recursively break down the problem into sub problems
2. Solve the sub problems in parallel
3. Combine the solutions to sub-problems to arrive at final result

## General Form

```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```



# Java library for Fork-Join framework

Started with JSR 166 efforts

JDK 7 includes it in `java.util.concurrent`

Can be used with JDK 6.

- Download jsr166y package maintained by Doug Lea
- <http://gee.cs.oswego.edu/dl/concurrency-interest/>



# Java library for Fork-Join framework

Started with JSR 166 efforts

JDK 7 includes it in `java.util.concurrent`

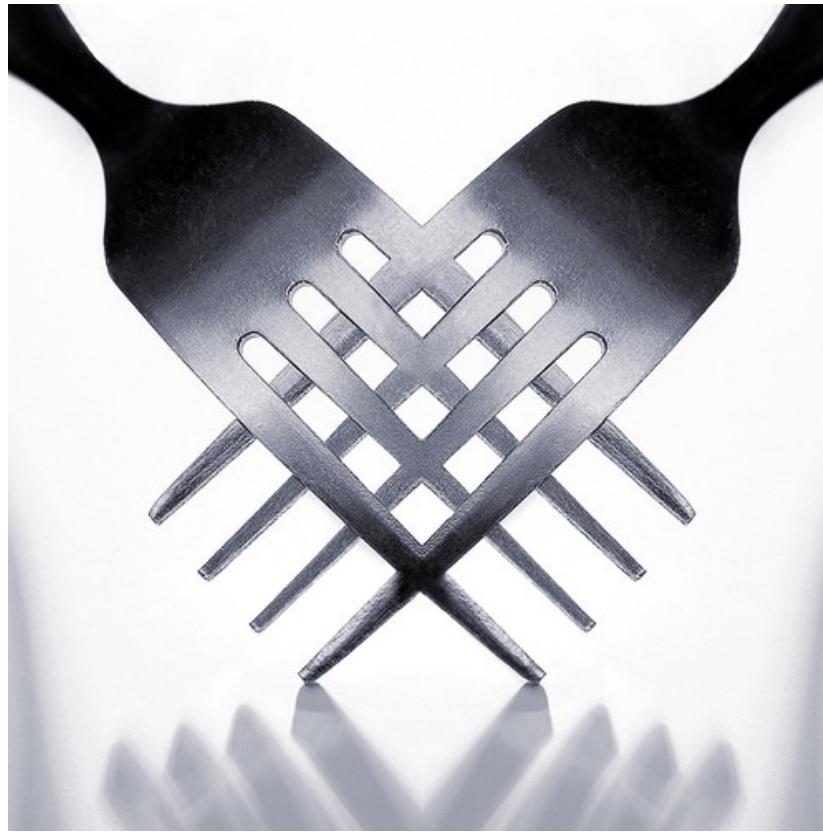
Can be used with JDK 6.

- Download jsr166y package maintained by Doug Lea
- <http://gee.cs.oswego.edu/dl/concurrency-interest/>



# Fork-Join framework

---



# Selecting a max number

```
public class SelectMaxProblem {  
    private final int[] numbers;  
    private final int start;  
    private final int end;  
    public final int size;  
  
    public SelectMaxProblem(int[] numbers2, int i, int j) {  
        this.numbers = numbers2;  
        this.start = i;  
        this.end = j;  
        this.size = j - i;  
        System.out.println("start:" + start + ",end:" + end + ",size:" + size);  
    }  
  
    public int solveSequentially() {  
        int max = Integer.MIN_VALUE;  
        for (int i=start; i<end; i++) {  
            int n = numbers[i];  
            if (n > max)  
                max = n;  
        }  
        System.out.println("returning max:" + max);  
        return max;  
    }  
  
    public SelectMaxProblem subproblem(int subStart, int subEnd) {  
        return new SelectMaxProblem(numbers, start + subStart,  
                                    start + subEnd);  
    }  
}
```

## Sequential algorithm



# Selecting max with Fork-Join framework

```
13 public class MaxWithForkJoin extends RecursiveAction {  
14     private final int threshold;  
15     private final SelectMaxProblem problem;  
16     public long result;  
17  
18     public MaxWithForkJoin(SelectMaxProblem problem, int threshold) {  
19         this.problem = problem;  
20         this.threshold = threshold;  
21     }  
22  
23     @Override  
24     protected void compute() {  
25  
26         if (problem.size < threshold) {  
27             result = problem.solveSequentially();  
28         }  
29         else {  
30             int midpoint = problem.size / 2;  
31  
32             MaxWithForkJoin left = new MaxWithForkJoin(problem.subproblem(0, midpoint), threshold);  
33             MaxWithForkJoin right = new MaxWithForkJoin(problem.subproblem(midpoint + 1, problem.size), threshold);  
34  
35             invokeAll(left, right);  
36  
37             result = Math.max(left.result, right.result);  
38         }  
39     }  
40  
41     }  
42  
43 }  
^
```

**A Fork-Join Task**

**Solve sequentially if problem is small**

**Otherwise Create sub tasks & Fork**

**Join the results**



# Fork-Join framework implementation

---

## Basic threads (Thread.start() , Thread.join())

- May require more threads than VM can support

## Conventional thread pools

- Could run into thread starvation deadlock as fork join tasks spend much of the time waiting for other tasks

ForkJoinPool is an optimized thread pool executor (for fork-join tasks)



# Fork-Join framework implementation

---

## Basic threads (Thread.start() , Thread.join())

- May require more threads than VM can support

## Conventional thread pools

- Could run into thread starvation deadlock as fork join tasks spend much of the time waiting for other tasks

ForkJoinPool is an optimized thread pool executor (for fork-join tasks)



---

ForkJoinTask as is could improve performance...

However some boilerplate work is still needed. Is there a better way?

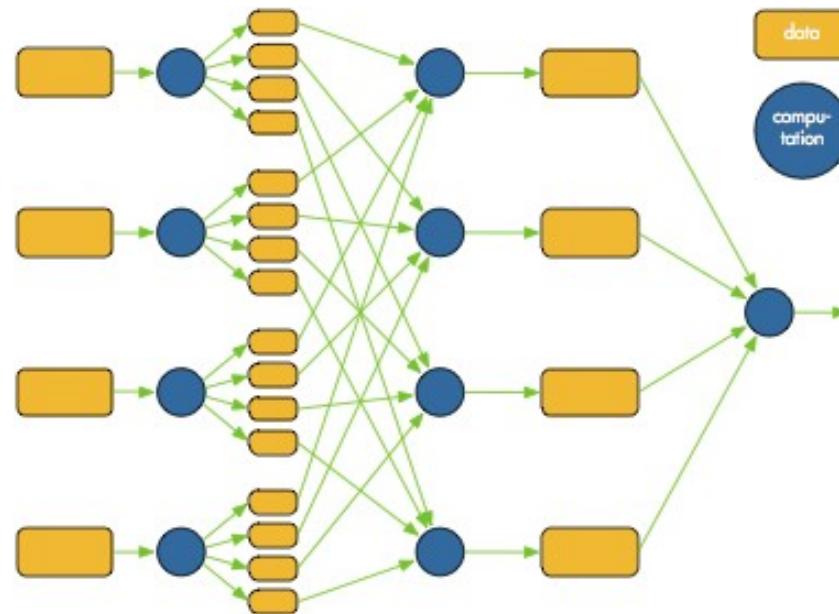
i.e. declarative specification of parallelization



# Options for Transparent Parallelism

## ParallelArray

- ForkJoinPool with an array
- Versions for primitives and objects
- Provide parallel aggregate operations
- Can be used on JDK 6+ with extra166y package from <http://gee.cs.oswego.edu/dl/concurrency-interest/>



# Example: ParallelArray Operations

```
30 ForkJoinPool forkJoinPool = new ForkJoinPool(Runtime.getRuntime().availableProcessors());
31 SLAData[] testData = getTestData(arrayLength);
32 ParallelArray<SLAData> slaData = ParallelArray.createUsingHandoff(testData, forkJoinPool);
33
34 Predicate<SLAData> june2013 = new Predicate<SLAData>() {
35     public boolean op(SLAData s) {
36         return s.year == 2013 && s.month == 6;
37     }
38 };
39
40 Predicate<SLAData> peakHour = new Predicate<SLAData>() {
41     public boolean op(SLAData s) {
42         return s.hour == 14;
43     }
44 };
45 Predicate<SLAData> deleteCommand = new Predicate<SLAData>() {
46     public boolean op(SLAData s) {
47         return s.command.equalsIgnoreCase("DEL-COMMAND");
48     }
49 };
50 Ops.ObjectToDouble<SLAData> selectResponseTime = new Ops.ObjectToDouble<SLAData>() {
51     public double op(SLAData s) {
52         return s.responseTime;
53     }
54 };
55
56 ParallelDoubleArray slaDataProcessed = slaData.withFilter(june2013)
57     .withFilter(peakHour )
58     .withFilter(deleteCommand)
59     .withMapping(selectResponseTime )
60     .all();
61 double average = slaDataProcessed.sum() / slaDataProcessed.size();
62
63 System.out.println("June 2013 DEL-COMMAND Average response time during peak hours:" + average );
64
65
66 double june2012PeakHourDeleteMaxResponseTime = slaData.withFilter(june2013)
67     .withFilter(deleteCommand)
68     .withMapping(selectResponseTime )
69     .max();
70
71 System.out.println("June 2013 DEL-COMMAND Max response time:" + june2012PeakHourDeleteMaxResponseTime);
```

ParallelArray  
created with  
**ForkJoinPool**

A filter  
operation

A map  
operation

SQL like..  
Batches  
operations in a  
single parallel  
step

Select max is  
a single step

# Lambda's make life easier..

```
Predicate<SLAData> june2013 = new Predicate<SLAData>() {
    public boolean op(SLAData s) {
        return s.year == 2013 && s.month == 6;
    }
};

Predicate<SLAData> peakHour = new Predicate<SLAData>() {
    public boolean op(SLAData s) {
        return s.hour == 14;
    }
};
Predicate<SLAData> deleteCommand = new Predicate<SLAData>() {
    public boolean op(SLAData s) {
        return s.command.equalsIgnoreCase("DEL-COMMAND");
    }
};
Ops.ObjectToDouble<SLAData> selectResponseTime = new Ops.ObjectToDouble<SLAData>() {
    public double op(SLAData s) {
        return s.responseTime;
    }
};

ParallelDoubleArray slaDataProcessed = slaData.withFilter(june2013)
    .withFilter(peakHour )
    .withFilter(deleteCommand)
    .withMapping(selectResponseTime )
    .all();

double average = slaDataProcessed.sum() / slaDataProcessed.size();
```

With Lambda's

```
slaData.parallelStream().filter(s -> s.getYear() == 2013)
    .filter(s -> s.getMonth() == JUNE)
    .filter(s -> s.hour() == 14)
    .filter(s -> s.getCommand() == DELETE)
    .mapToInt(s -> s.getResponseTime())
    .sum();
```

# Fork-Join and Map-Reduce

|                          | Fork-Join                                                      | Map-Reduce                                |
|--------------------------|----------------------------------------------------------------|-------------------------------------------|
| Processing on            | Cores on a single compute node                                 | Independent compute nodes                 |
| Division of tasks        | Dynamic                                                        | Decided at start-up                       |
| Inter-task communication | Allowed                                                        | No                                        |
| Task redistribution      | Work-stealing                                                  | Speculative execution                     |
| When to use              | Data size that can be handled in a node                        | Big Data                                  |
| Speed-up                 | Good speed-up according to #cores, works with decent size data | Scales incredibly well for huge data sets |

Source: <http://www.macs.hw.ac.uk/cs/techreps/docs/files/HW-MACS-TR-0096.pdf>





## Issues Parallel Stream



## Streams

---

- What is a stream?
  - A stream is a ~~collection sequence stream~~ of ~~objects~~.
  - A stream is an **abstraction** that represents zero or more **values**.
- Not (necessarily) a collection: values might not be stored anywhere
- Not (necessarily) a sequence: order might not matter
- Values, not objects: avoid mutation and side effects

## Pipelines

- A pipeline consists of:
  - a stream *source*
  - zero or more *intermediate* operations
  - a *terminal* operation

```
collection.stream()      // source
              .filter(...)    // intermediate operation
              .map(...)        // intermediate operation
              .collect(...);   // terminal operation
```



## Parallel Streams

- Source starts with stream(), parallelStream(), or other stream factory
- Can be switched using parallel() or sequential() calls
- Parallel vs sequential is a property of the entire pipeline
  - can't switch between parallel and sequential in the middle
  - “last one wins”
- Parallel makes it auto-magically go faster, right?

```
collection.stream()  
    .filter(...)  
    .parallel()  
    .map(...)  
    .sequential()  
    .collect(...);
```

// entire stream runs sequentially

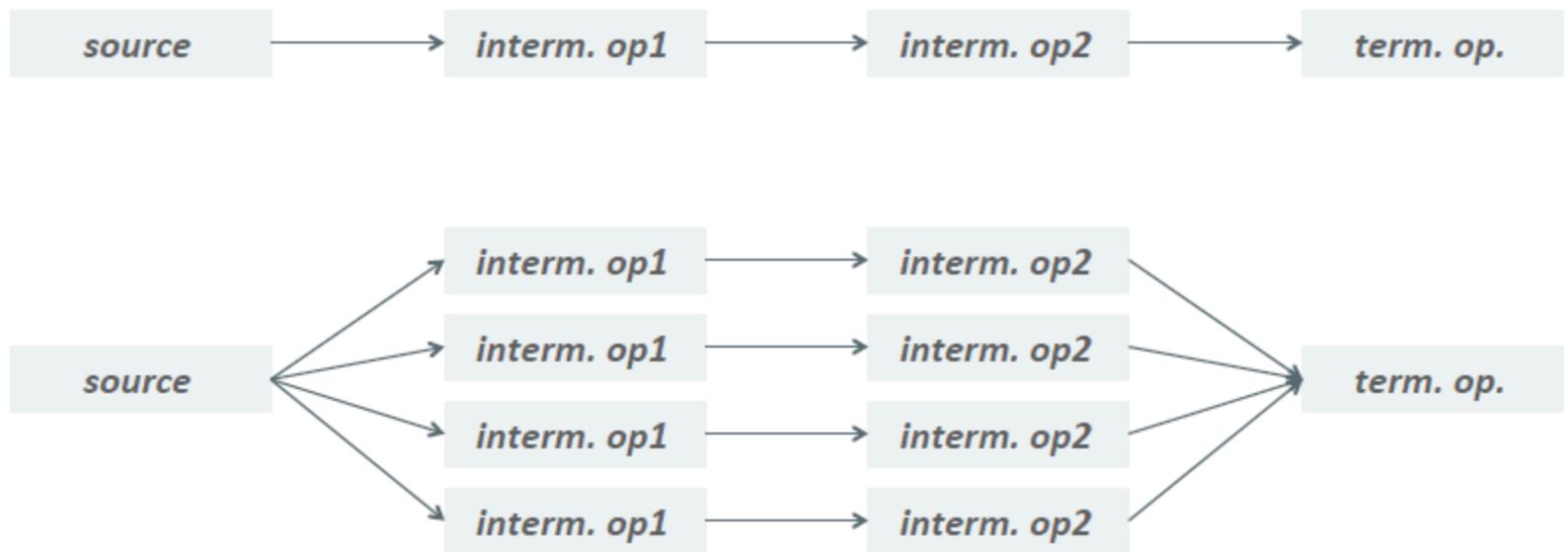
---

## Parallel Streams Considerations

- Parallel and sequential streams should give the same result
  - parallelism leads to nondeterminism, usually bad! Need to control it.
- Encounter order vs. processing order
- Stateless vs. stateful: managing side effects
- Accumulation vs. Reduction
- Reduction: identity and associativity
- Explicit nondeterminism can speed things up
- Parallelism has overhead, might slow things down



## Sequential vs. Parallel Streams



## Sequential and Parallel Streams

```
List<String> output = IntStream.range(0, 50)
    .filter(i -> i % 5 == 0)
    .mapToObj(i -> String.valueOf(i / 5))
    .collect(toList());
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]



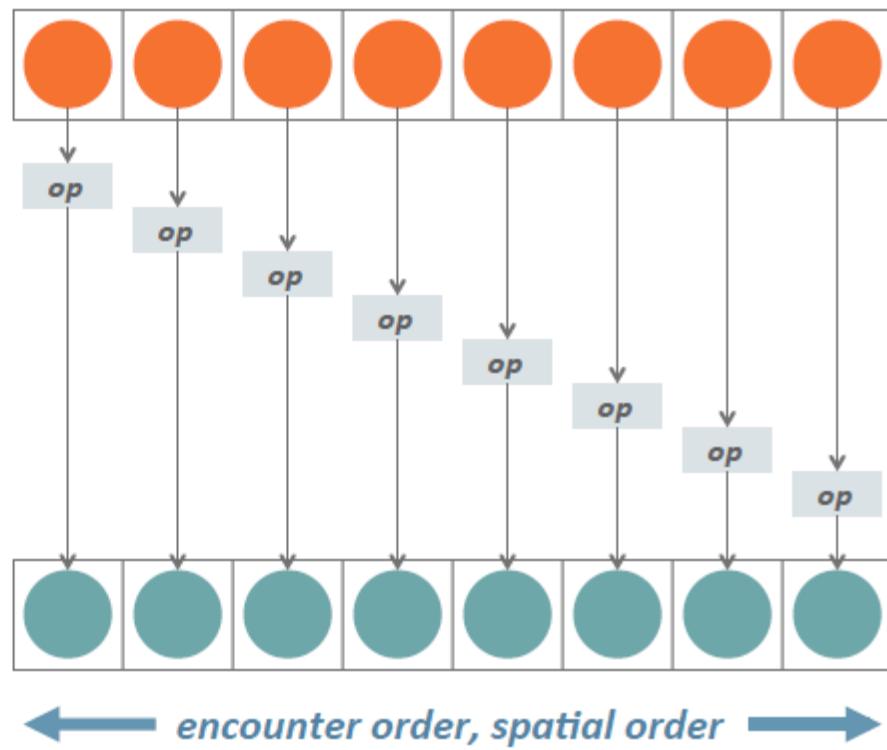
```
List<String> output = IntStream.range(0, 50)
    .parallel()
    .filter(i -> i % 5 == 0)
    .mapToObj(i -> String.valueOf(i / 5))
    .collect(toList());
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

*Same result; how  
is this possible?*

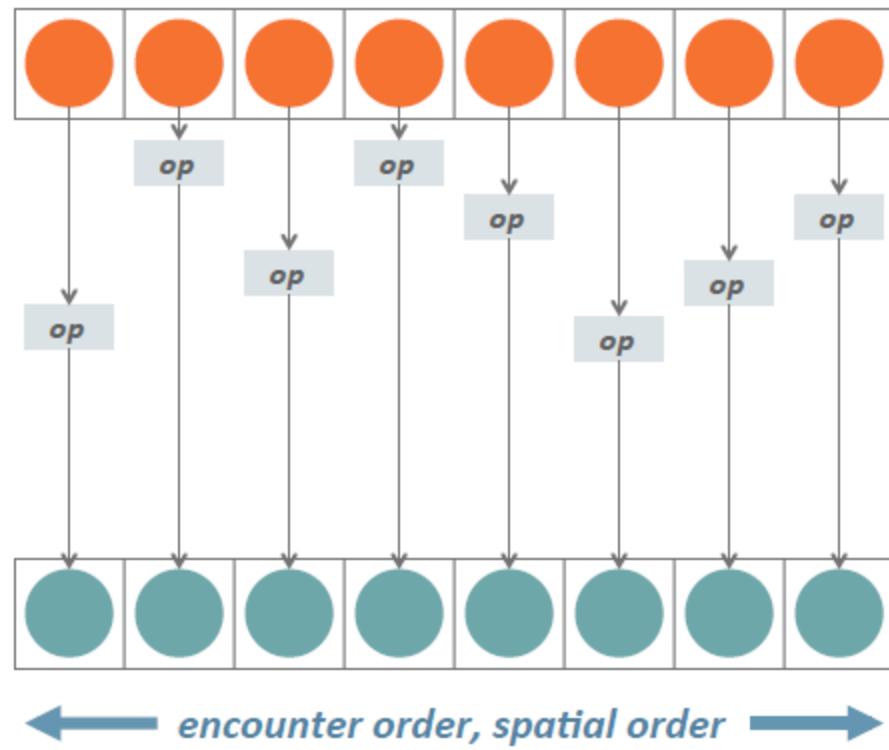
## Ordering Sequential

*Time*  
↓  
*processing order, temporal order*



## Ordering Parallel

*Time*  
*processing order  
temporal order*



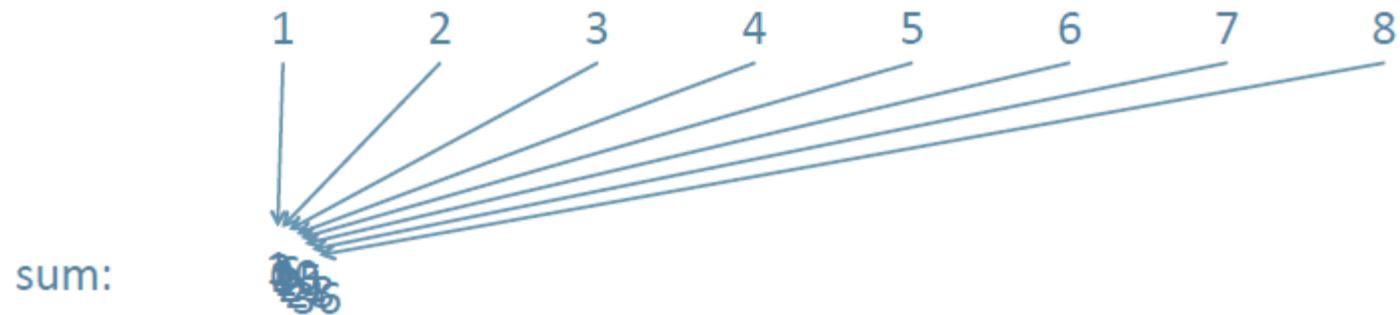
---

## Accumulation vs. Reduction

```
long sum = 0L;  
  
for (long i = 1L; i <= 1_000_000L; i++) {  
    sum += i;  
}  
  
System.out.println(sum);  
  
500000500000
```



## Summation by Accumulation



*Contention!*



## A Better Way: Reduction

1

2

3

4

5

6

7

8



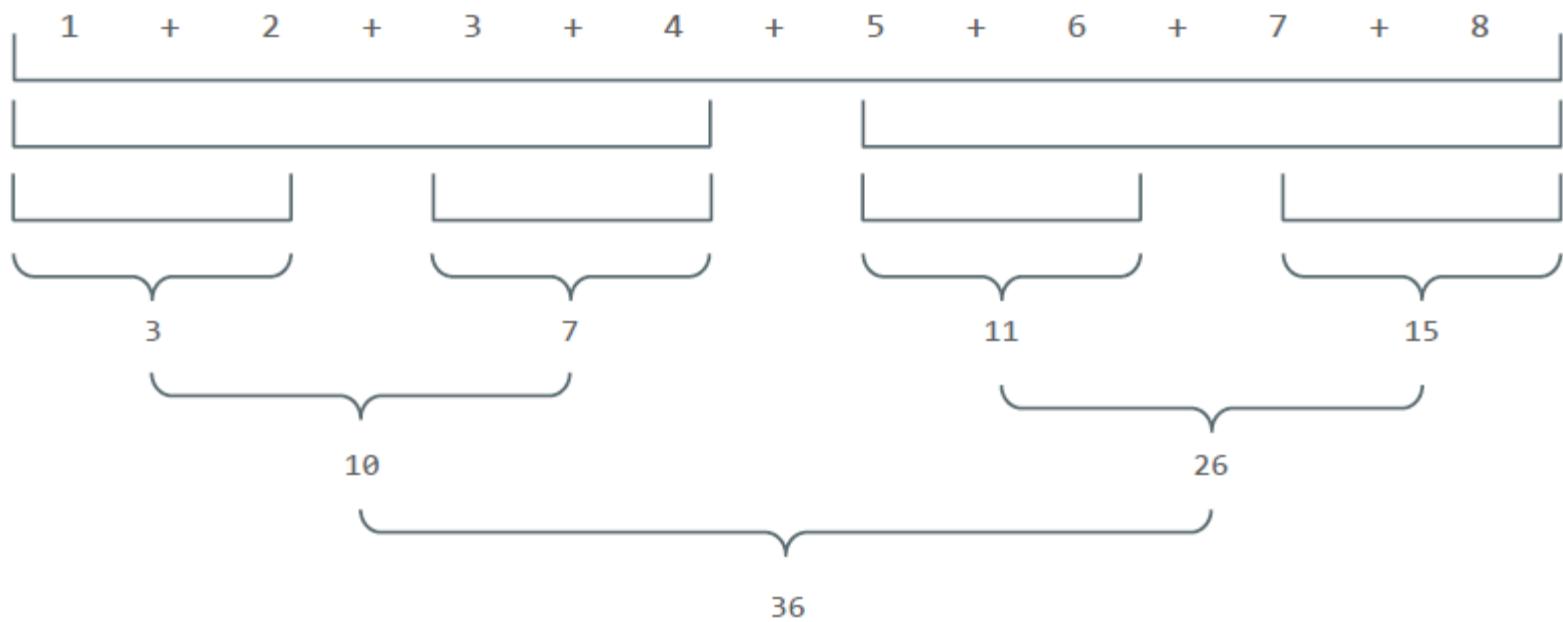
## A Better Way: Reduction

1 + 2 + 3 + 4 + 5 + 6 + 7 + 8

*Reduction over addition:  
Just put a plus between each value.*



## Reduction Implementation



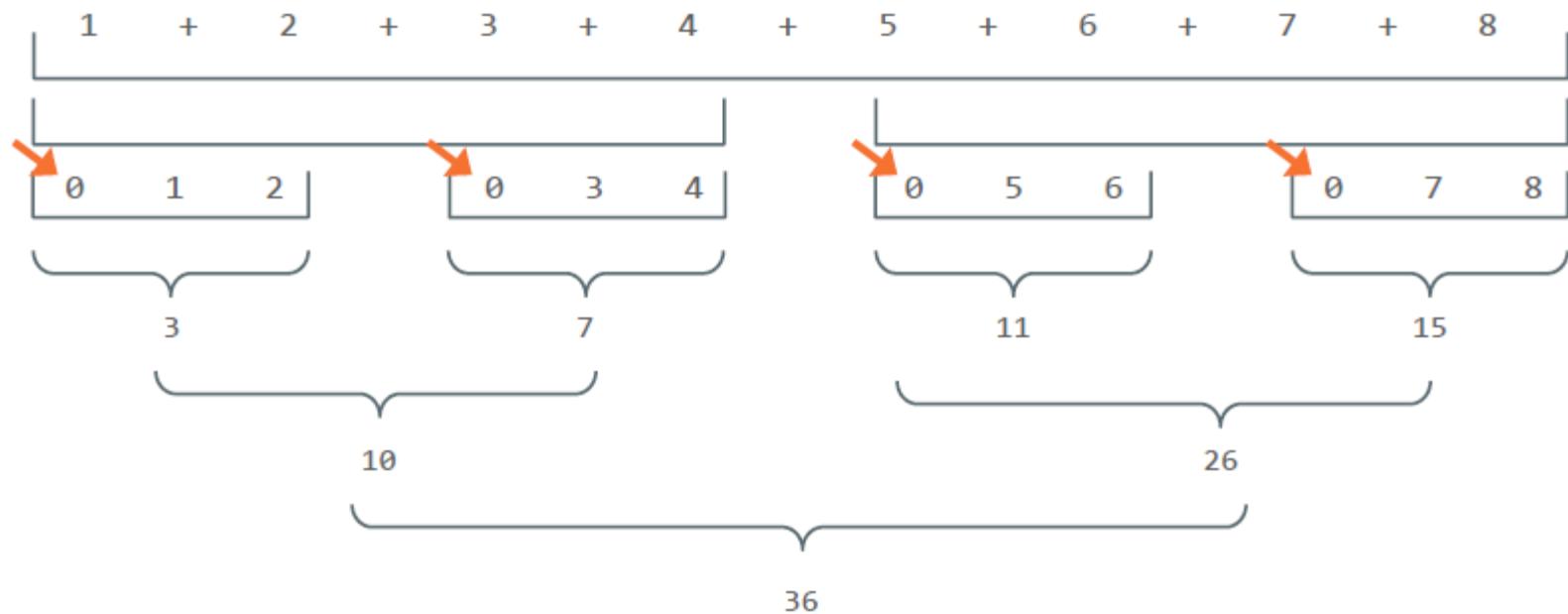
---

## Reduction – Identity

- Identity Value
  - the starting value of each partition of a parallel reduction
  - becomes the result if there are no values in the stream
  - it must be the *right* identity value
  - must really be an identity *value* (immutable, not mutable)



## Reduction Implementation



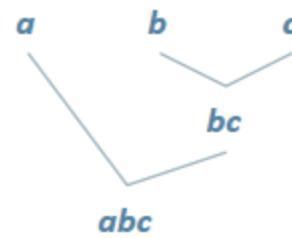
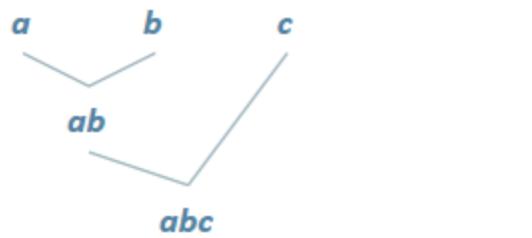
## Associativity

- Remember elementary arithmetic?  
–  $(a + b) + c = a + (b + c)$  ?
- Turns out it's quite important
- A function is *associative* if different groupings of operands don't affect the result
- Reduction functions must be associative



## Associativity

```
(String x, String y) -> x + y
```

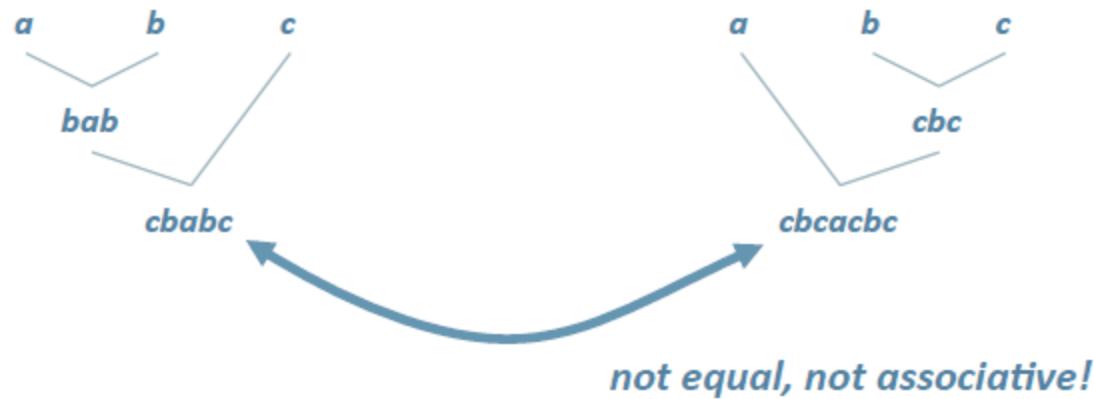


*equal: yes, associative*



## Associativity

(String x, String y)  $\rightarrow$  y + x + y



---

## Reduction: Summary

- Consider reduction in preference to accumulation
- Reduction can be subtle
  - rewriting an accumulation into a reduction can be non-obvious
- Identity must be an immutable value
- Reduction function must be associative



## Non-determinism

- Usually we want to get the same result for parallel as sequential
- Consider `findFirst()`
  - “first” means first in encounter (spatial) order
  - parallel can find a matching element quickly
  - but still has to search space to the left to ensure it’s first
- Consider `findAny()`
  - parallel can find a matching element quickly
  - and it’s done!



---

## Where are the Threads?

- Parallel stream initiated by `parallelStream()` or `parallel()` call
  - who starts the threads? where do they come from?
- Stream workload split and dispatched to the ***common fork-join pool***
- Control over concurrency explicitly opaque in the API
  - allocation of resources should be by administrator/deployer, not programmer
  - common FP pool controlled by system properties; needs to be enhanced
- Policy APIs need development
  - split policy, degree of parallelism, handling blocking tasks



**EXTRA**



# Project coin: Java 7

A set of small languages changes to  
simplify day  
to day programming task

1. Binary integral literals
2. Number with \_
3. String in switch
4. Type inference for generic creation
5. catching multiple exception types
6. Improved rethrowing exceptions
7. Resource management ARM (Try With Resources)



# Binary integral literals

- With java7 we can create numeric literals using binary notation "0b"
- There is no direct way to create binary literals up to java6

```
public class BinaryIntegerLiterals {  
  
    public static void main(String[] args) {  
        int n=0b1000000000000; //getting in binary  
        System.out.println(n);  
        byte data=0b111111; //up to 8 digit  
        data=0b01000001; //cant convert int to byte  
        data=0b000000001111; //leading zero ignored  
        data=1b0000001; //negative not by 1b  
        data=-0b000001100; //negative by -0b  
        long ldata=0b11111111111111111111111111L;  
        float f=0b00111111000;  
    }  
}
```

# number with underscore \_

- How easy to read this number?  
int k=100000000;

```
int k=100000000;
int n1=1_00000_0;
int n2=1_____000;
double b=1_0000_00.00_00;
```

- Consecutive underscore is not valid
- Underscore can be used with other data type also
- Underscore can be applied after underscore

# String in switch

```
String sport="hockey";
switch(sport){
    case "tennis":System.out.println("tennis");
                    break;
    case "volleyball":System.out.println("volleyball");
                        break;
    case "TT":System.out.println("TT");
                    break;

    default:System.out.println("cricket");
}

Object sport=null;
switch(sport){
    case "tennis":System.out.println("tennis");
                    break;
    case "volleyball":System.out.println("volleyball");
                        break;
    case "TT":System.out.println("TT");
                    break;

    default:System.out.println("cricket");
}
```

# Type inference for generic creation

- Earlier when using generics we have to specify the type twice
- Compiler can infer the type.....
- In Java 6
  - `List<Integer> list=new ArrayList<Integer>();`
- Now in Java 7
  - `List<Integer> list=new ArrayList<>();`

# Improved rethrowing exceptions

```
class FirstException extends Exception{}  
class SecondException extends Exception{}  
  
class Demo{  
  
    public static void rethrowException(String n) throws FirstException, SecondException  
    {  
        try{  
            if(n.equals("fist"))  
                throw new FirstException();  
            else  
                throw new SecondException();  
        }  
        catch(Exception e){throw e;}  
    }  
}  
  
public class ImprovedRethrowEx {  
    public static void main(String[] args) {  
        try {  
            Demo.rethrowException("feist");|  
        } catch (Exception e) {  
            System.out.println(e.getClass());  
        }  
    }  
}
```

- Compiler can determine that exception thrown by the statements must comes from try block and it can be either first exception of SecondException.

# Catching multiple exception types

```
class FirstException extends Exception{}  
class SecondException extends Exception{}  
  
public class CatchingMultipleExTypes {  
    public static void main(String[] args) {  
        try{  
            if("foo".equals("foo"))  
                throw new FirstException();  
            else  
                throw new SecondException();  
        }  
        catch(FirstException|SecondException e){  
            e.printStackTrace();  
        }  
    }  
}
```

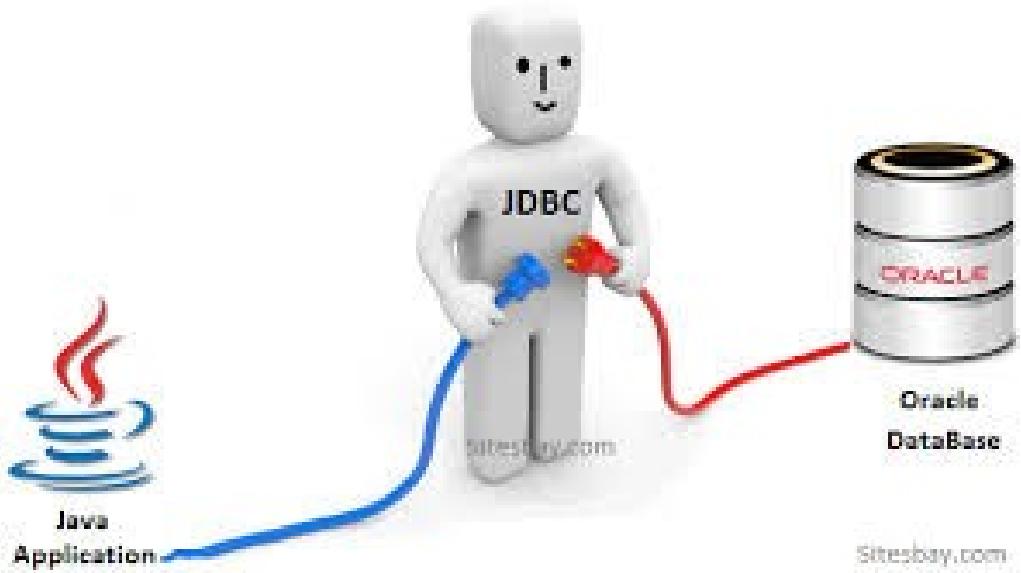
# Resource management ARM (Try With Resources)

- Prior to java7, resources such as JDBC connections, files ,IO streams should be closed by programmer manually.....
- What if we don't care :
  - Orphan instances of resources leads to inefficient resource management.
- Java7 gives <<java.lang.AutoClosable>> interface this interface has one method promise....close()

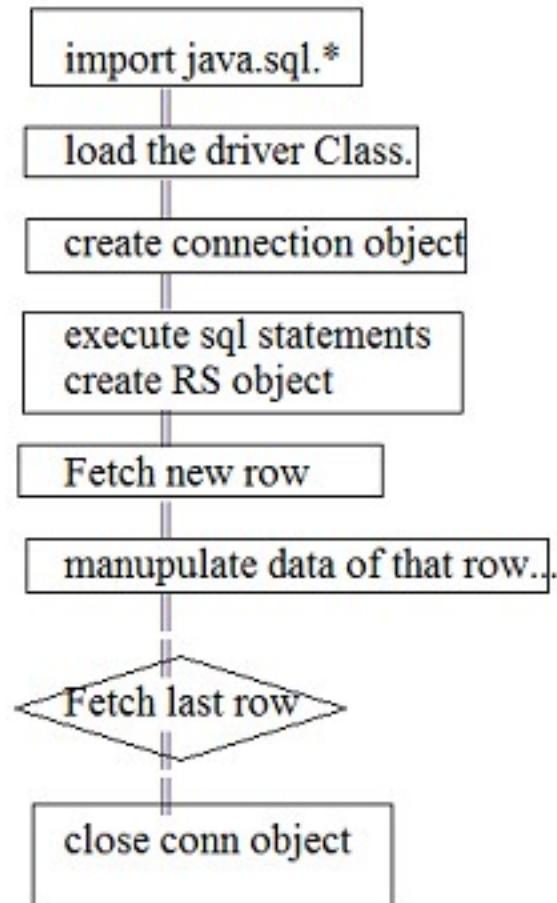
# Example

```
class OpenDoor implements AutoCloseable{
    @Override
    public void close() throws Exception {
        System.out.println("close method for OpenDoor is called by JVM.....");
    }
    public OpenDoor(){
        System.out.println("constructor of OpenDoor class.....");
    }
}

public class TWR1 {
    public static void main(String[] args) {
        try(
            OpenDoor d=new OpenDoor();
        )
        {
        }
        catch(Exception e){}
    }
}
```

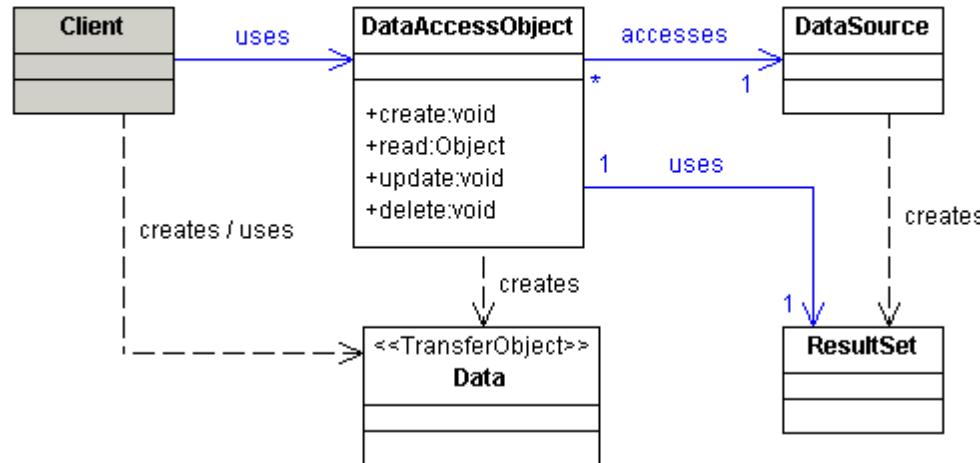


# JDBC



# DAO DTO design Pattern

## DAO / DTO design pattern





# Spring Webflux

- @Controller
- Functional API
- Combinators
- WebClient
- Backpressure
- Schedulers

## Simple web request flow

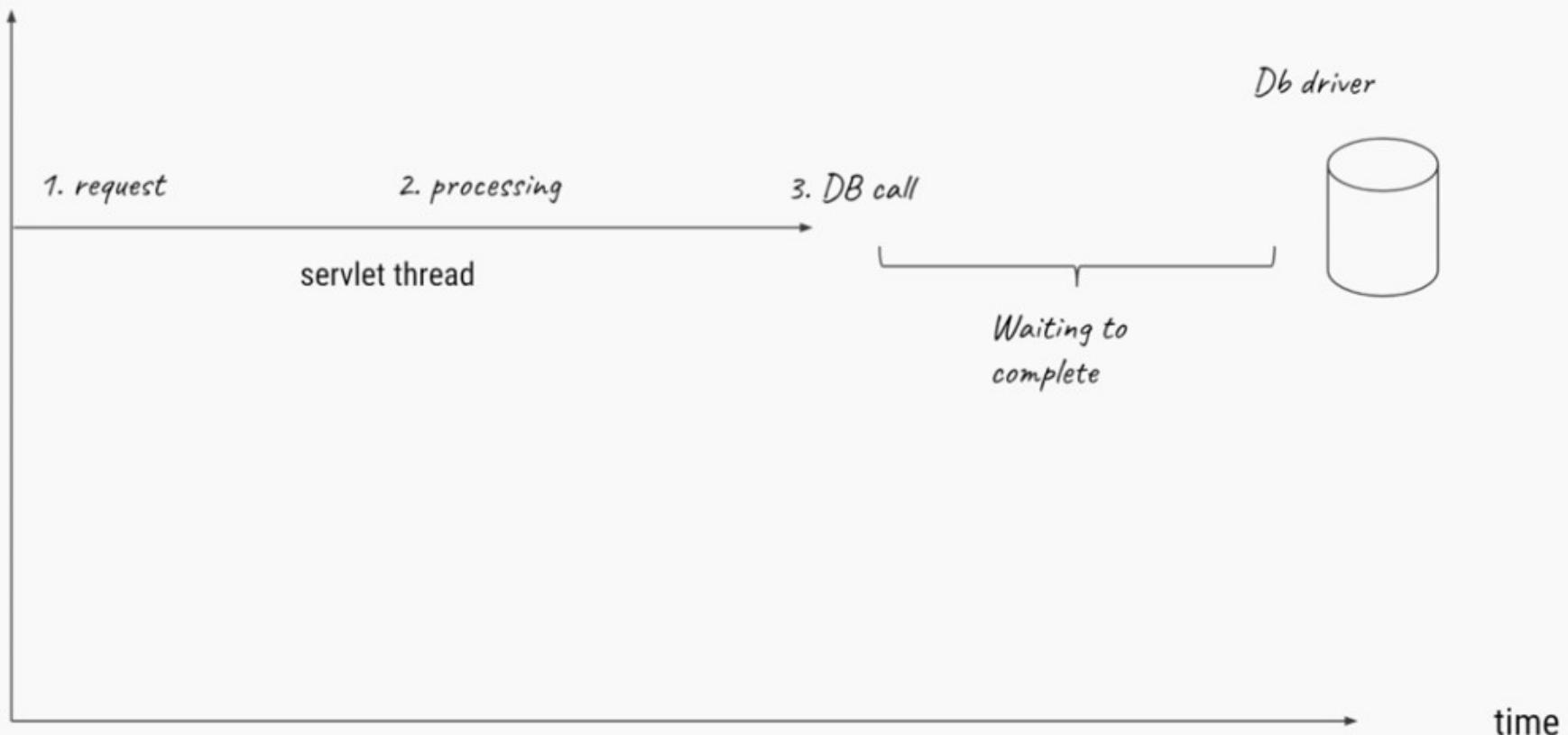
```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public User getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

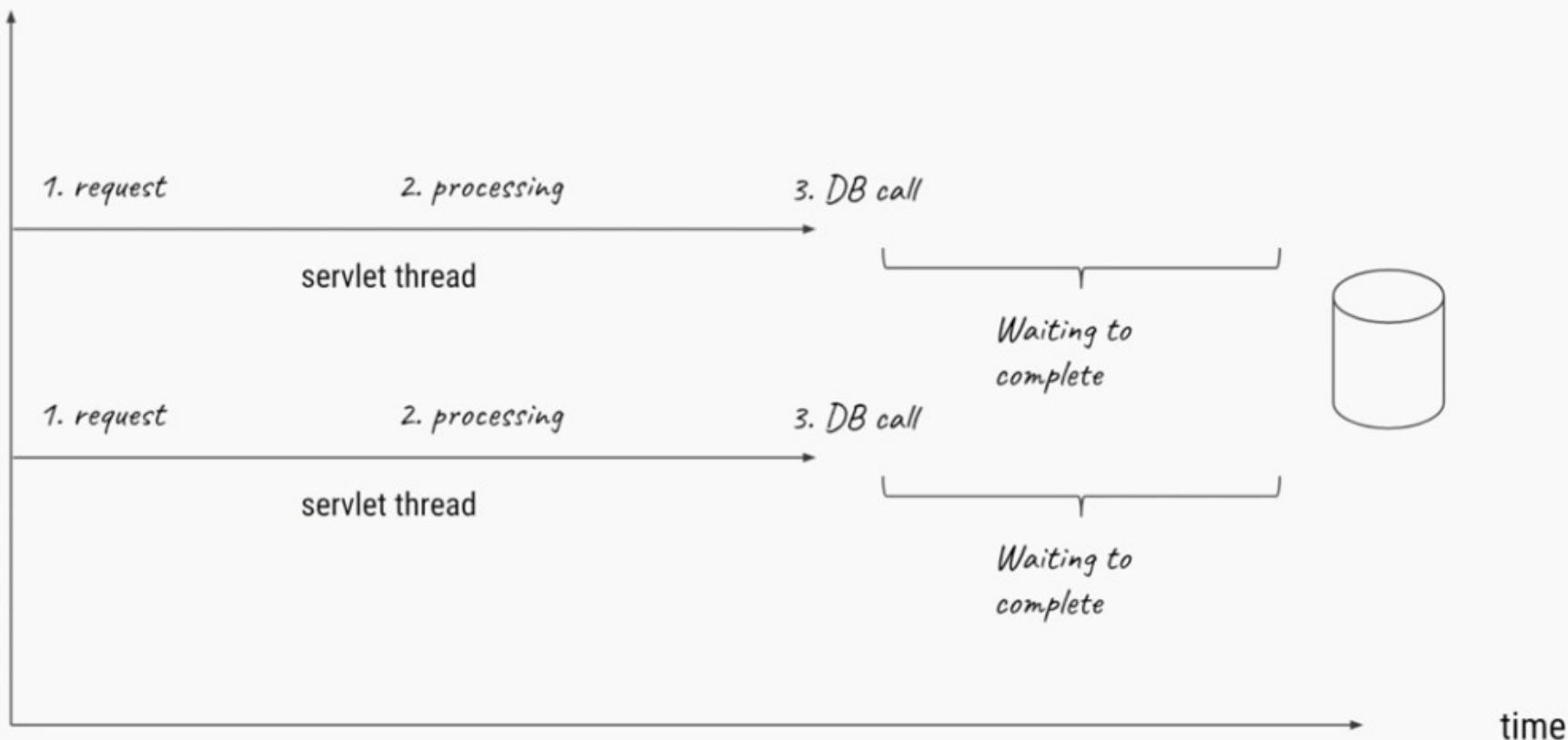
## Web requests with IO

threads



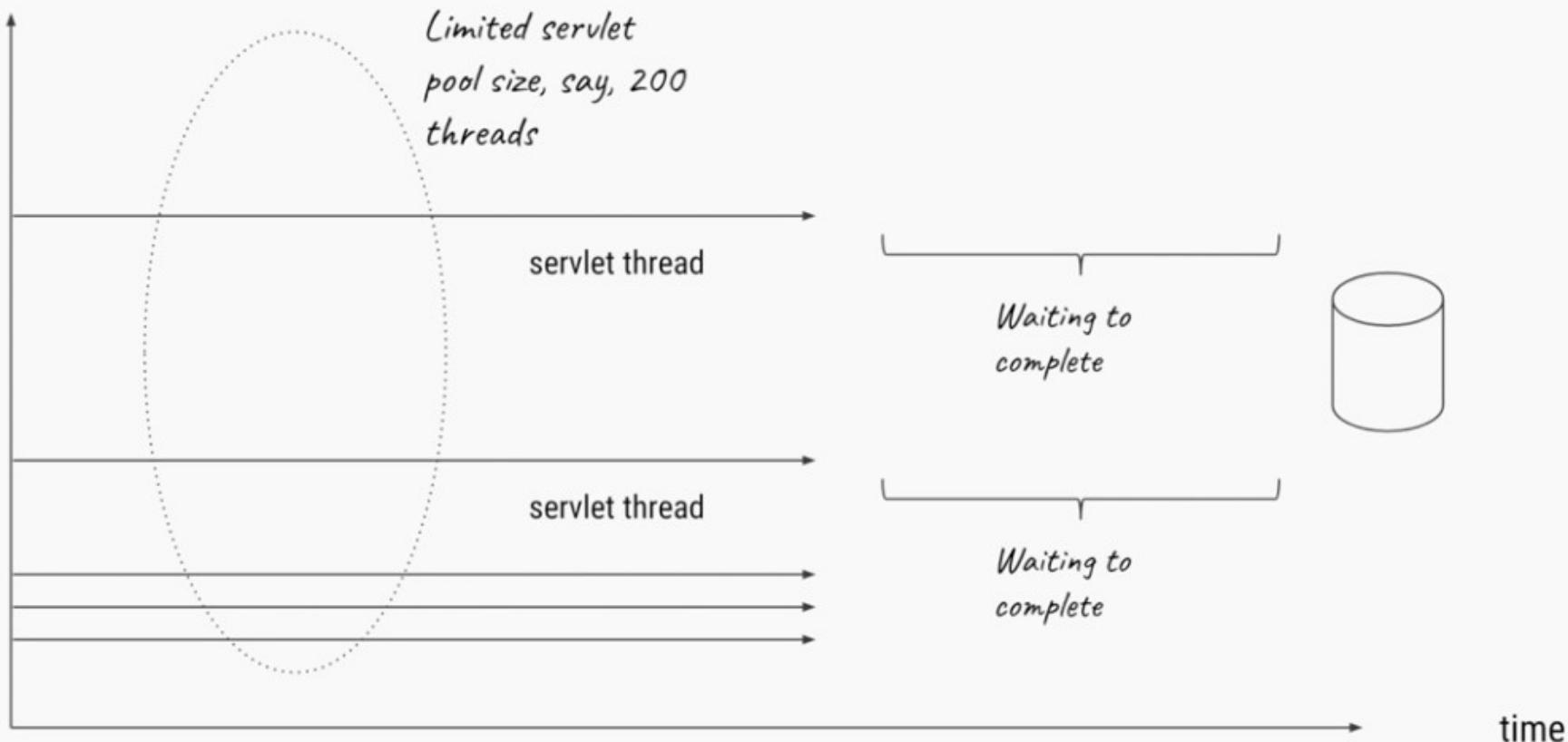
# Web requests with IO

threads

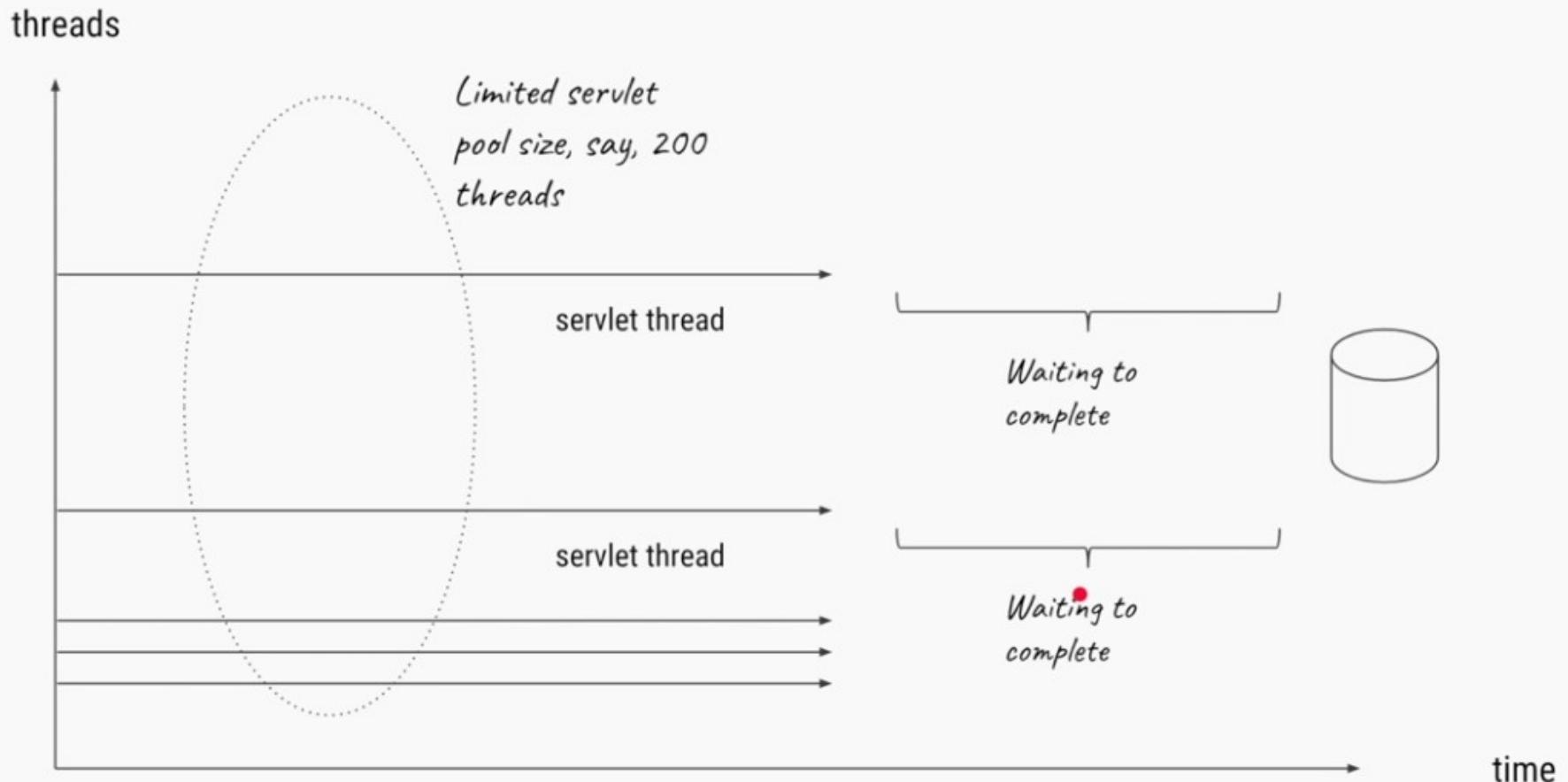


## Web requests with IO

threads

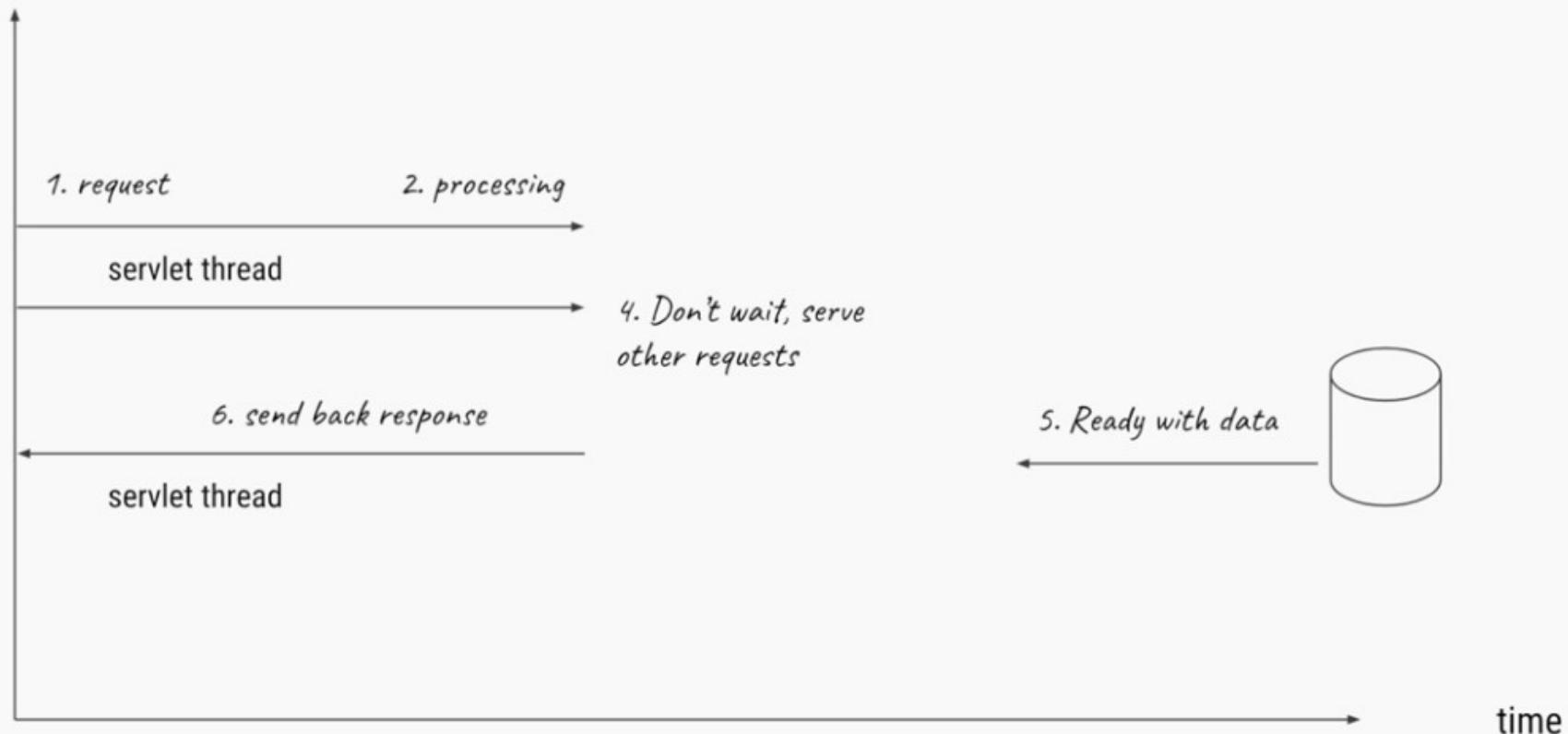


## Web requests with IO



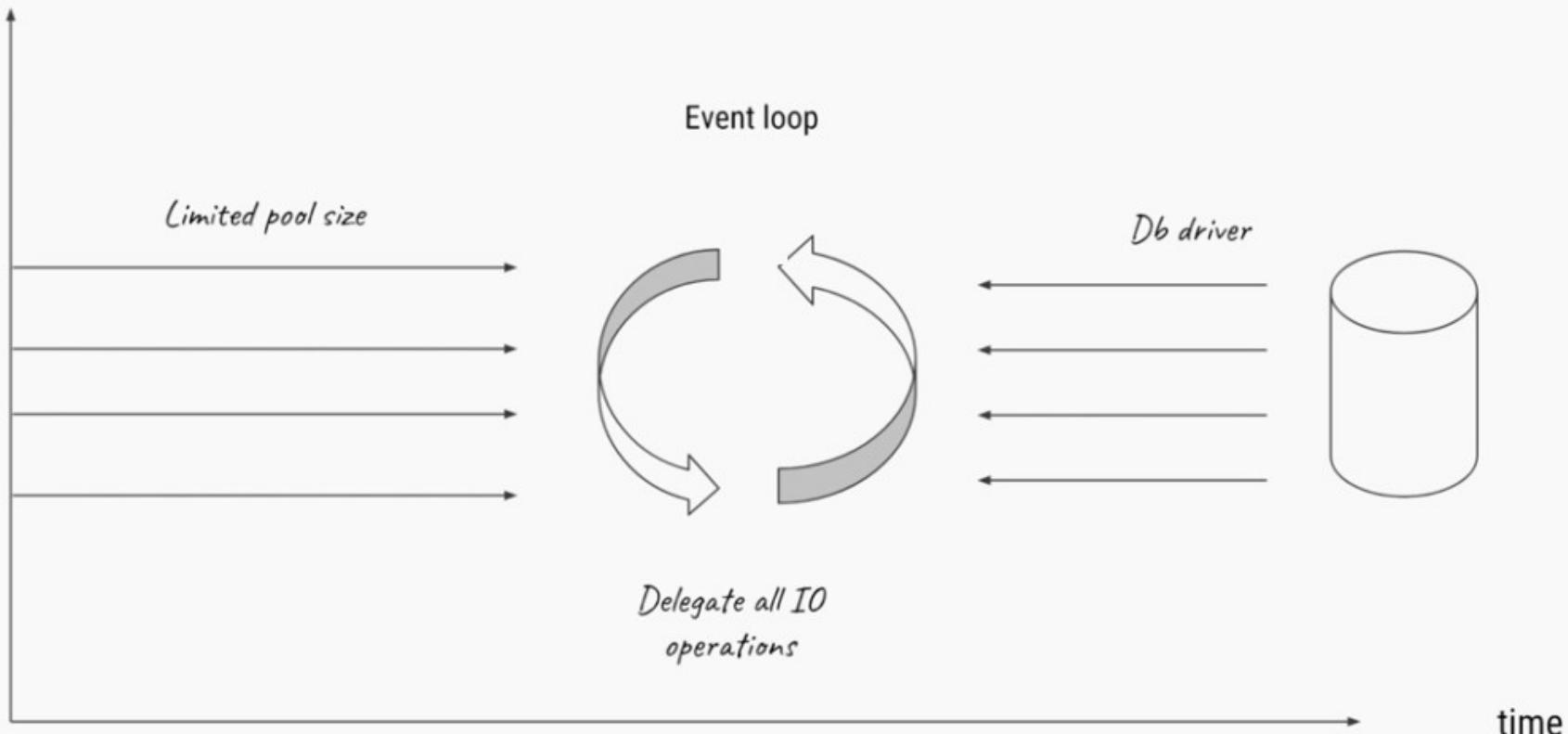
## Web requests with IO

threads



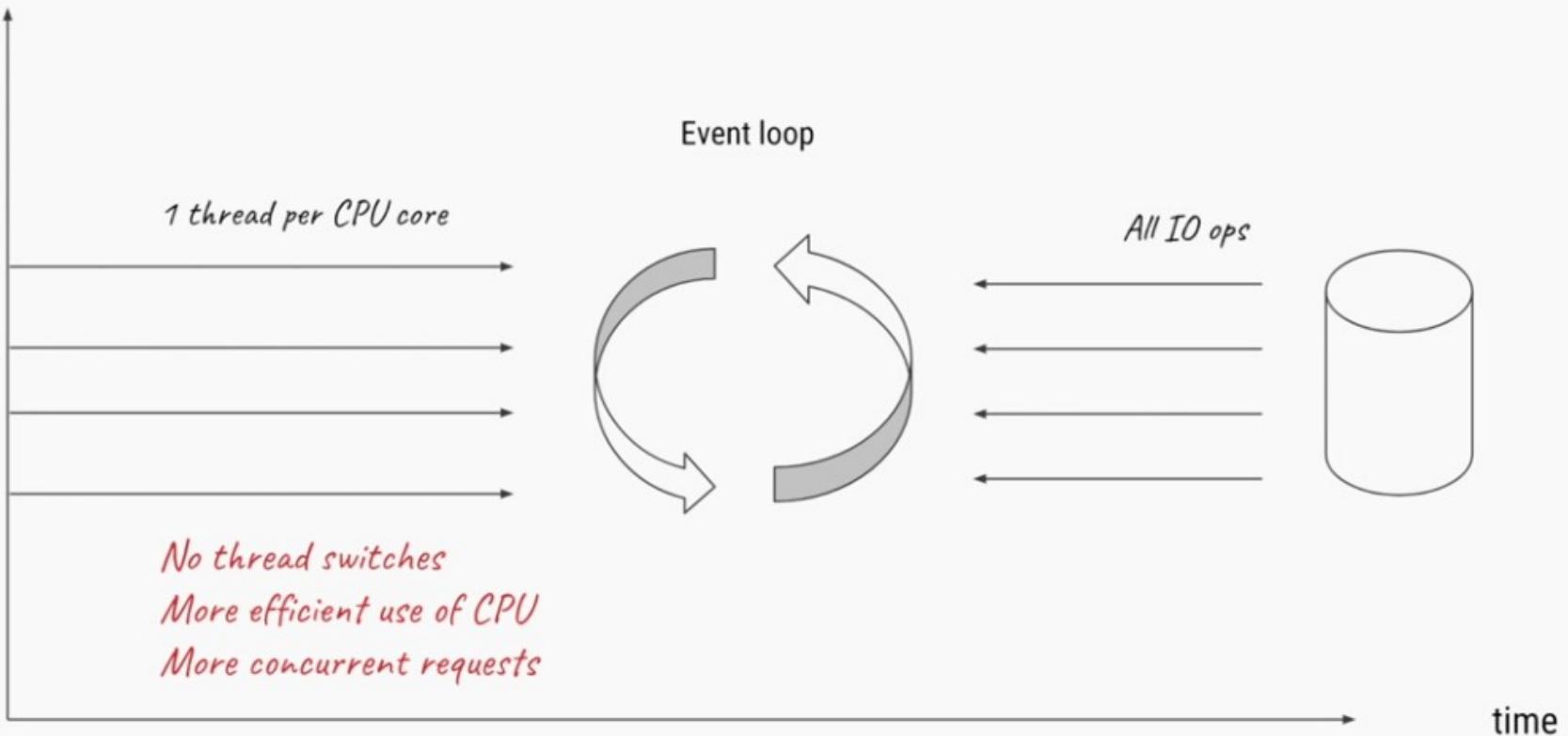
# High throughput

threads

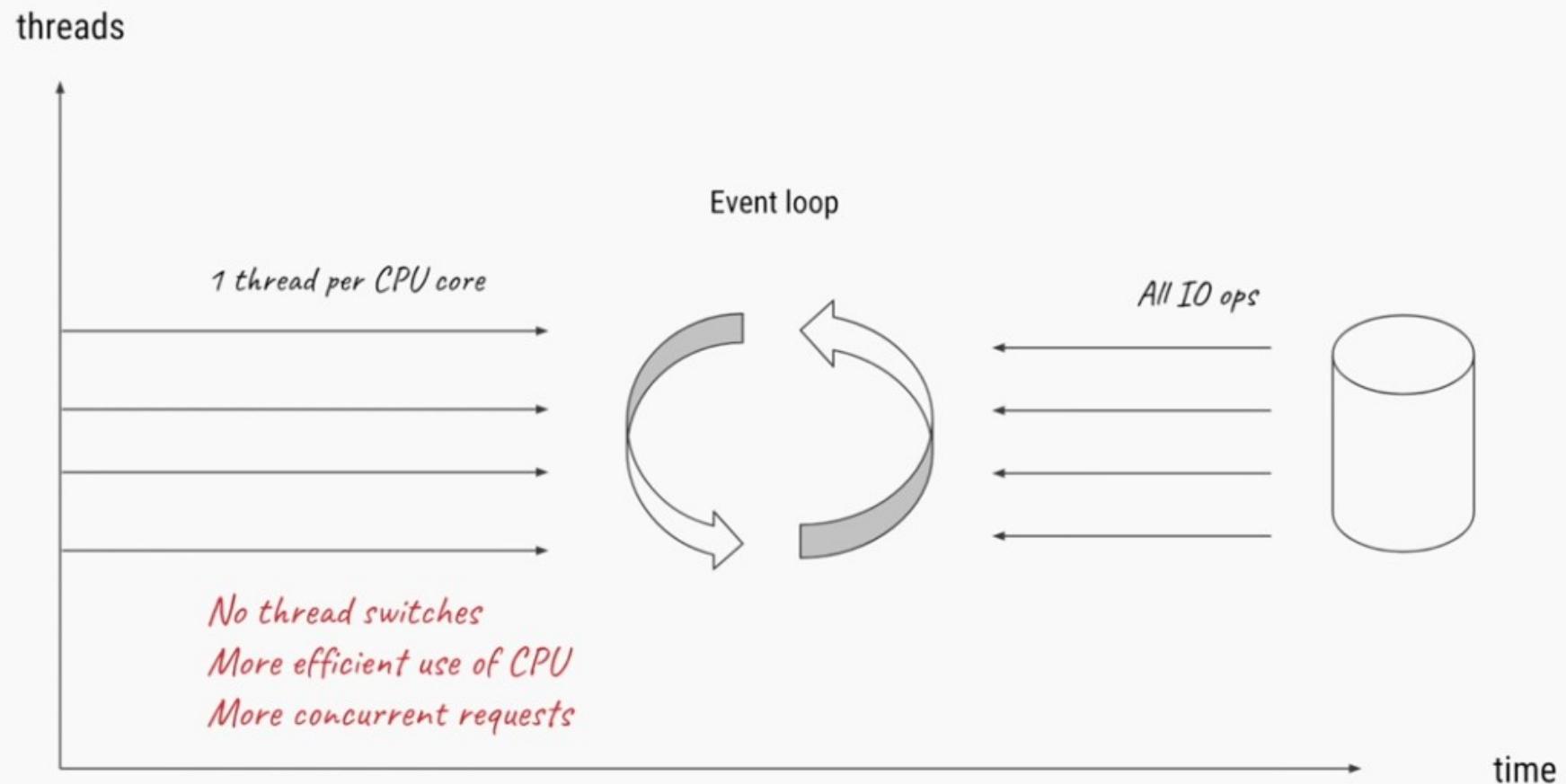


# High throughput

threads

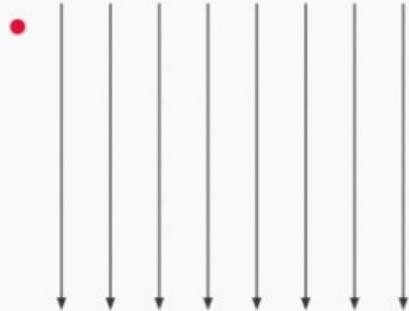


## High throughput



## Traditional vs Event Loop

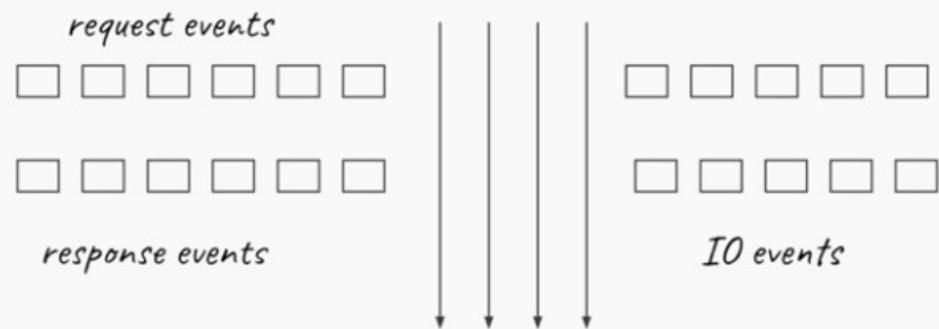
1 thread per request flow



|       |       |
|-------|-------|
| core1 | core2 |
| core3 | core4 |

Lot of thread switches  
Scheduling of threads

1 thread per CPU core

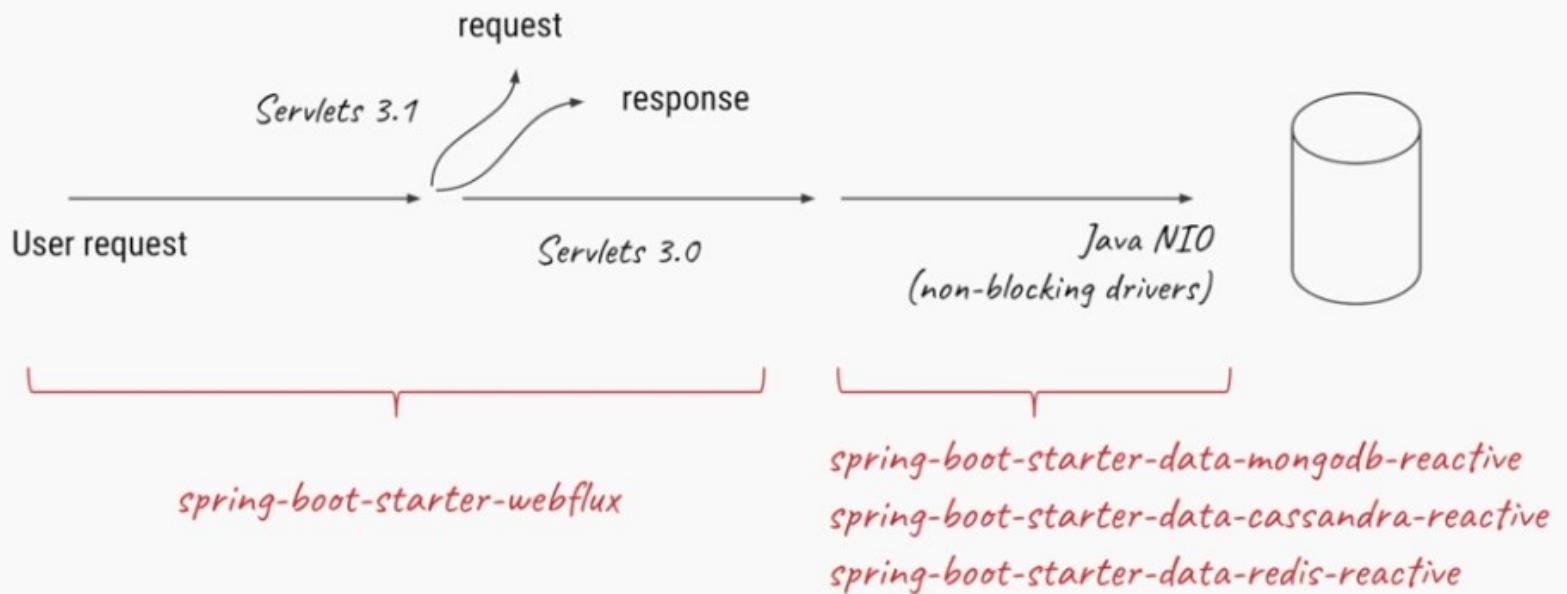


|       |       |
|-------|-------|
| core1 | core2 |
| core3 | core4 |

No thread switching  
Increased complexity

## End to end reactive

- Servlets 3.0 and 3.1 (Async and NIO)
- Java NIO (File, Network IO)
- MongoDB, Cassandra, Redis (more to come)



## Simple web request flow

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userrepository;

    @GetMapping("/get/user/{id}")
    public User getUser(@PathVariable String id) {
        return userrepository.findUser(id); •
    }
}
```

*Blocking call*

Future as placeholders?

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public Future<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

## CompletableFuture as placeholders?

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public CompletableFuture<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

## Spring Webflux - using Reactor

```
@RestController
public class MyRestController {

    @Autowired
    private UserRepository userRepository;

    @GetMapping("/get/user/{id}")
    public Mono<User> getUser(@PathVariable String id) {
        return userRepository.findUser(id);
    }
}
```

*Mono = 0..1 elements*

*Flux = 0..N elements*

# Hot/Live sources of data

## Continuous data responses

```
@RestController
public class PricingController {

    @Autowired
    private PricesRepository pricesRepository; Reactive DB driver

    @GetMapping("/get/prices/{id}")
    public Mono<Price> getPrice(@PathVariable String id) {
        return pricesRepository.findById(id); Single value
    }

    @GetMapping("/get/prices/all")
    public Flux<Price> getAllPrice() {
        return pricesRepository.findAll(); List of values
    }

    @GetMapping(value = "/get/prices/live", produces = "text/event-stream")
    public Flux<Price> getLivePrice() {
        return pricesRepository.findAll(); Live list of values
    }
}
```

## Continuous data responses



## Also works with requests

```
@RestController
public class PricingRequestsController {

    @Autowired
    private PricesRepository pricesRepository; Reactive DB driver

    @PostMapping("/save/price")
    public void save(@RequestBody Mono<Price> price) {
        price.subscribe(pricesRepository::save);
    } Single value

    @PostMapping("/save/price/all")
    public void saveAll(@RequestBody Flux<Price> prices) {
        pricesRepository.saveAll(prices);
    } List of values

    @PostMapping(value = "/save/price", consumes = "application/stream+json")
    public void getPrice(@RequestBody Flux<Price> prices) {
        pricesRepository.saveAll(prices);
    } Live list of values
}
```

## Continuous data requests



# When to use?

## Advantages

- Scalability
  - IO bound operations (microservices)
  - Efficient CPU utilization
  - Data locality & Less-context switches
- Streaming use-cases (Live source of data)
- Backpressure

## Considerations

- Different programming model
- End-to-end reactive required
- Not too useful for CPU bound flows
- Hard to debug (stack trace)
- Hard to write tests