Rishabh Indoria
21F3001823

Programming Concepts using Java

October 3, 2024

1 Introduction

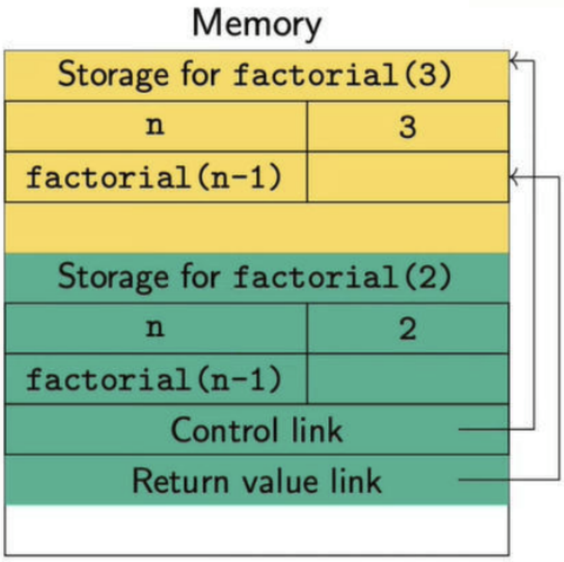
1.1 Types

- **The role of types**: Interpreting data stored in binary consistently, i.e., viewing a sequence of bits as integers, floats, characters etc., Naming concepts and structuring out computations, Catching bugs early.
- **Dynamic Typing**: Derive type from the current value, $x = 10$ means x is of type int.
- **Static Typing**: Associate a type in advance with a name, $int\ x$.
- In static typing x will remain int for its lifetime, i.e., cannot assign float or char value to x whereas this is possible in dynamic typing. **Static analysis**: With variable declarations, compilers can detect type errors at compile time.
- Whereas dynamic typing would catch these errors only when the code runs.
- Compilers can also perform optimizations based on static analysis.

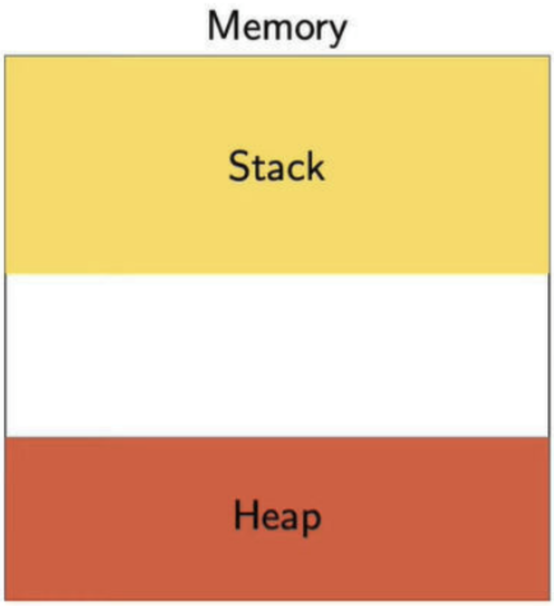
1.2 Memory Management

- Variables store intermediate values during computation.
- Typically these values are local to a function but can also refer to global variables outside the function as well.
- **Scope** of a variable means when the variable is available to use.
- **Lifetime** of a variable is how long the storage remains allocated.
- "Hole in the scope": Variable is alive but not is scope.
- **Memory stack**: Each function needs storage for local variables. Create **an activation record** when function is called.
- Activation record are stacked which is popped when the function exits.
- **Control link** points to start of previous record.
- **Return value link** tells where to store the result.
- When a function is called, arguments are substituted for formal parameters.
- Parameters are part of the activation record of the function. Values are populated on function call. Like having implicit assignment statements at the start of the function.
- **Call by value**: Copy the value, updating the value inside the function has no side effect.
- **Call by reference**: parameter points to same location as the argument, updating the value will have side effects.
- Need a separate storage for persistent data, usually called the heap.
- Conceptually, allocate heap storage from "opposite" end with respect to the stack.
- Heap store outlives the activation record whereas in stack, variables are deallocated when a function exits.
- After deleting a node in a linked list, deleted node is now dead storage, unreachable, requires some memory management.
- **Manual Memory Management**: Programmer explicitly requests and returns heap storage. This is error-prone, memory leaks, invalid arguments.

- **Automatic Garbage Collection**: Run-time environment checks and cleans up dead storage. Marks all storage that is reachable from program variables and return all unmarked memory cells to free space. Convenience for programmer vs performance penalty.



(a) Activation Record Example



(b) Heap Memory

1.3 Stepwise Refinement

- Begin with a high level description of the task.
- Refine the tasks into subtasks, and further elaborate each subtask.
- Subtasks can be coded by different people.
- **Program refinement**: focus on code, not much change in data structures.

1.4 Modular Software Development

- Use refinement to divide the solution into components. Build a prototype of each component to validate design.
- Components are described in terms of interfaces and specification.
- **Interfaces**: What is visible to other components, typically function calls.
- **Specification**: Behavior of the component, as visible through the interface.
- Improve each component independently, preserving interface and specification.

1.5 Programming Language support for Abstraction

- **Control Abstraction**: Functions and procedures. Encapsulate a block of code, reuse in different contexts.
- **Data Abstraction**: Abstract data types(ADT), set of values along with operations permitted on them. Internal representation should not be accessible, interaction should be restricted to public interface.
- **Object-Oriented Programming**: Organize ADTs in a hierarchy. Implicit reuse of implementations, subtyping, inheritance.

2 Object-Oriented Programming

2.1 Classes and Objects

- An object is like an abstract data type. Hidden data with set of public operations.
- All interactions are done through operations.

- Uniform way of encapsulating different combinations of data and functionality.
- Classes are a template for a data type, i.e., how a data is stored and how public functions manipulate the data.
- Objects are concrete instances of the above mentioned template. Each object maintains its separate copy of local data. Invoking methods on objects is equivalent to "send a message to the object".
- Point example in python

```python
class Point:
    def __init__(self, a=0, b=0):
        self.x = a
        self.y = b
    def translate(self, dx, dy):
        self.x += dx
        self.y += dy
    def odistance(self):
        import math
        d = math.sqrt(self.x*self.x + self.y*self.y)
        return d
```

- We can change the internal implementation from (x, y) to (r, θ).

2.2 Abstraction

- Objects are similar to abstract data types. They have public interface, private implementation, and changing the implementation should not affect interactions with the object.
- Data-centric view of programming, focus on what data we need to maintain and manipulate.
- Refining data representation is naturally tied to updating methods that operate on the data.
- Users of the code should not know whether *Point* uses (x, y) or (r, θ) implementation.

2.3 Subtyping

- We can arrange types in a hierarchy.
- A subtype is a specialization of a type.
- If A is a subtype of B, then wherever an object of type B is needed, an object of type A can be used.
- Every object of type A is also an object of type B. Think of subsets, if $X \subseteq Y$, every $x \in X$ is also in Y.
- If $f()$ is a method in B and A is a subtype of B, every object of A also supports $f()$, although implementation of $f()$ can be different in A.

2.4 Dynamic lookup

- Whether a method can be invoked on an object is a static property, aka **type checking**.
- How the method acts is a dynamic property of how the object is implemented.
- Invoke the same operation, each object "knows" which function to invoke.
- Different from **overloading**.
- A variable v of type B can refer to an object of subtype A.
- Static type of v is B, but method implementation depends on runtime type A.

2.5 Inheritance

- Reuse of implementations.
- Usually one hierarchy of types to capture both subtyping and inheritance.
- A can inherit from B iff A is a subtype of B.
- Philosophically, the two are different.
- Subtyping is a relationship of interfaces whereas Inheritance is a relationship of implementations.

3 Java

3.1 First Taste of Java

- Let's start with printing "hello world".

```
public class helloworld{
    public static void main(String[] args){
        System.out.println("hello, world");
    }
}
```

- All code in Java lies within a class. No free floating functions.
- Modifier *public* specifies visibility.
- For the code to start, it requires the *main* function with the correct signature.
- *static* makes it, so the function can be run independent of objects.
- *System* is a public class, *out* is a **stream** object defined in *System*. *println*() is a method associated with streams.
- A Java program is a collection of classes. Each class is defined in a separate file with the same name, with extension *.java*.
- Java programs are usually interpreted on **Java Virtual Machine (JVM)**.
- *javac* compiles into JVM **byte code**. *java* runs the code.
- *javac* should be provided *.java* extension and *java* should not be provided *.class*.

3.2 Data Types

- Java has 8 primitive data types. Size of each is fixed by JVM independent of native architecture.

Type	Size in bytes
int	4
long	8
short	2
byte	1
float	4
double	8
char	2
boolean	1

Figure 2: Java Basic Data Types

- We need to declare variables before we use them.
- Characters are written with **single-quotes**, strings are marked with **double-quotes**.
- Boolean constants are *true*, *false*.
- Modifier *final* marks a constant. This variable cannot be updated later.

- Arithmetic operators are the usual ones $+, -, *, /, \%$
- When both arguments are integer, then / is integer division.
- There is no exponentiation operator, use $Math.pow(a, n) = a^n$.
- *String* is a built-in class. Strings are **not** an array of characters.

```
String s = "Hello", t = "world";
String u = s + "-" + t; // "Hello world"
s = s.substring(0,3) + "p!";
int length = s.length();
```

- If we change a *String*, we get a new object. *String* are **immutable**.
- Arrays are also objects

```
int[] a;
a = new int[100];
int[] b = new int[100];
int length = a.length;
a = {2, 3, 5, 7, 11};
```

- In arrays, *length* is a variable, whereas in strings *length*() is a method.
- Java allows limited **type inference** only for local variables in a function and not for instance variables in a function.
- Use generic *var* to declare variables, these must be initialized when declared. Their type is inferred from initial value.

3.3 Control Flow

- Start with conditional statements

```
public class MyClass{
    ...
    public static int sign(int v){
    if (v < 0)
        return(-1);
    else if (v > 0)
        return(1);
    else
        return(0);
    }
}
```

- Conditional Loops

```
public class MyClass{
    ...
    public static int sumupto(int n){
        int sum = 0;
        while (n > 0){
            sum += n;
            n--;
        }
        return(num);
    }
}
```

- Conditional loop can also be *do...while(c)*;
- Iterative loop

```java
public class MyClass{
    ...
    public static int sumarray(int[] a){
        int sum = 0;
        int n = a.length;
        for(int i = 0; i < n; i++){
            sum += a[i];
        }
        return(sum);
    }
}
```

- Java later introduced a *for* in the style of Python

```java
for(type x : a){
    do something with x;
}
```

- Multiway branching aka switch case statement.

```java
public static void printsign(int v){
    switch(v){
        case -1:{
            System.out.println("Negative");
            break;
        }
        case 1:{
            System.out.println("Positive");
            break;
        }
        case 0:{
            System.out.println("Zero");
            break;
        }
        default:{
            System.out.println("Fuck, Something went wrong");
        }
    }
}
```

3.4 Defining Classes and Objects

- Definition block using *class*, with class name. Default visibility is public to *package*.
- All classes defined in the same directory from part of the same package.
- **Instance variable**: Each concrete object type *Date* will have local copies of $date, month, year$. These are marked *private*. Can also have *public* instance variable, but breaks encapsulation.
- Declare type using class name, *new* creates a new object.
- *this* is a reference to current object. We can omit *this* is reference is unambiguous.
- We can add accessor and mutator methods to access and modify private variables.
- Constructors: Special functions called when an object is created. We can create multiple constructors with different signatures.
- A later constructor can call an earlier one using *this* keyword.
- **Copy constructor** takes an object of the same type as an argument.
- If instance variables are objects, we may end up aliasing rather than copying aka shallow copy.
- An example of above concepts

```java
public class Date{
    private int day, month, year;
    public Date(int d, int m, int y){
        day = d;
        month = m;
        year = y;
    }
    public Date(int d, int m){
        this(d, m, 2021);
    }
    public Date(Date d){
        this.day = d.day;
        this.month = d.month;
        this.year = d.year;
    }
    public int getDay(){
        return(day);
    }
    public int getMonth(){
        return(month);
    }
    public int getYear(){
        return(year);
    }
}

public void useDate(){
    Date d1, d2;
    d1 = new Date(12, 4, 1954);
    d2 = new Date(d1);
}
```

3.5 Input and Output

- We already saw an example of output.
- For input, the easiest to use is the *Console* class, defined within *System*.

```java
Console cons = System.console();
String username = cons.readLine("Username: ");
char[] password = cons.readPassword("Password: ");
```

- A more general *Scanner* class, allows more granular reading of the input.

```java
Scanner in = new Scanner(System.in);
String name = in.nextLine();
int age = in.nextInt();
```

- Output can be done with *System.out*.
- *println*(*arg*) prints *arg* and goes to a new line.
- *print*(*arg*) prints *arg* but doesn't advance to a new line.
- *printf*(*arg*) generates formatted output, same convention as in *C* language.

3.6 Class Hierarchy

- Java does not allow multiple inheritance, it is tree like.
- In fact, there is a universal superclass *Object*.
- Few useful methods define in *Object*.

```java
public boolean equals(Object 0) // defaults to pointer equality
public String toString() /* converts the value of
instance variables to String */
```

- For Java object x and y, $x == y$ invokes $x.equals(y)$
- Example of overriding *equals* with *Date*

```
public boolean equals(Object d){
    if(d instanceof Date){
        Date myd = (Date) d;
        return this.day == myd.day && this.month == myd.month &&
        this.year == myd.year;
    }
    return false;
}
```

- Overriding looks for "closest" match.

3.7 Modifiers

- *public* vs *private* to support encapsulation of data.
- *static*, for entities defined inside classes that exist without creating objects of the class.
- $final$, for values that cannot be changed.

3.8 Abstract Classes and Interfaces

- Provide an abstract definition of the method

```
public abstract double perimeter();
```

- Cannot create objects from a class that has abstract functions, and the class containing an abstract function must be declared abstract.
- Abstract class forces subclasses to provide a concrete implementations for abstract methods.
- We can still declare variable whose type is an abstract class.
- We can use abstract classes to specify generic properties.

```
public abstract class Comparable{
    public abstract int cmp(Comparable s);
    // return -1 if this < s
    // return 0 if this == s
    // return 1 if this > s
}
```

- Now we can sort any array of objects that extend *Comparable*.
- An interface is an abstract class with no concrete components.
- An interface is an abstract class with no concrete components.
- We can extend only one class, but can implement multiple interfaces.
- Interface are basically classes with all methods being abstract.

```
public interface Comparable{
    public abstract int cmp(Comparable s);
}

public class Circle extends Shape implements Comparable{
    public double perimeter(){...}
    public int cmp(Comparable s){...}
    ...
}
```

- Interface describes relevant aspects of a class. Abstract functions describe specific "slice" of capabilities.
- Java interfaces extended to allow functions to be added. We can provide a default implementation for some functions.

- If there is conflict between static/default methods then subclass must provide a fresh implementation.
- Conflict could be between a class and an interface, then the class "wins".
- **Functional interfaces**: Interfaces that define a single function. Examples can be Comparator, TimerOwner.
- We can use lambda expression to replace these

```
Arrays.sort(strarr, (String s2, String s2) -> s1.length - s2.length)
```

- More complicated function body can be defined as a block.
- We can reference static and instance methods as follows

```
(x1, x2, ..., xk) -> f(x1, x2, ..., xk)
(o, x1, x2, ..., xk) -> o.f(x1, x2, ..., xk)
```

3.9 Private Classes

- *LinkedList* is built using *Node*. Why should *Node* be public?
- Make *Node* a private class nested within *LinkedList*, also called an **inner class**.
- Objects of private class can see private components of enclosing class.

3.10 Controlled interaction with Objects

- Take the example of querying a database.
- Object stores train reservation information. We need to control spamming by bots, do this by requiring the user to login before querying.
- Need to connect the query to the logged in status of the user. Use objects, on login return a Query object.
- How does the user know the capabilities of the private object?, use an interface.
- Query object could allow unlimited number of queries. Limit the number of queries per login by maintaining a counter.

```
public interface QIF{
    public abstract int getStatus(int trainno, Date d);
}

public class RailwayBooking{
    private BookingDB, railwayDB;
    public QIF login(String u, String p){
        QueryObject qobj;
        if(validLogin(u,p)){
            qobj = new QueryObject();
            return qobj;
        }
    }

    private class QueryObject implements QIF{
        private int numQueries;
        private final int QLIM;
        public int getStatus(int trainno, Date d){
            if(numQueries < QLIM){
                // Respond, increment numQueries
            }
        }
    }
}
```

3.11 Callbacks

- Myclass creates a Timer, and start it to run in parallel.
- Timer would notify Myclass when the time limit expires.
- Interface Runnable indicates that Timer can run in parallel.

```
public interface TimerOwner{
    public abstract void timerDone();
}

public class MyClass implements TimerOwner{
    public void f(){
        ...
        Timer t = new Timer(this); // this object create t
        ...
        t.start(); // Start t
        ...
    }
    public void timerDone(){...}
}

public class Timer implements Runnable{
    // Timer can be invoked in parallel
    private TimerOwner owner;
    public Timer(TimerOwner o){
        owner = o; // Creator
    }
    public void start(){
        ...
        owner.timerDone();
    }
}
```

3.12 Iterators

- We want to loop to run through all the values in a list, but we do not have public access, and we do not know which implementation it uses.
- Need the following abstraction

```
Start at the beginning of the list
while(there is a next element){
    get the next element;
    do something with it
}
```

- Create an *Iterator* object and export it

```
public interface Iterator{
    public abstract boolean has_next();
    public abstract Object get_next();
}

public class LinearList{
    private class Iter implements Iterator{
        private Node position;
        public Iter(){...}
        public boolean has_next(){...}
        public Object get_next(){...}
    }
    // Export a fresh iterator
    public Iterator get_iterator(){
        Iter it = new Iter();
```

```
            return it;
        }
    }
```

- Definition of *Iter* depends on the linear list.
- For nested loops. acquire multiple iterators.

3.13 Generics Programming

- Use type variables. Type quantifier before return type.
- "For every type T ..."
- Polymorphic *reverse* in Java

```
public <T> void reverse(T[] arr){
    T temp;
    int n = arr.length;
    for(int i = 0; i < n/2; i++){
        temp = arr[i];
        arr[i] = arr[(n - 1) - i];
        arr[(n - 1) - i] = temp;
    }
}
```

- The type parameter T can also be applies to the class as a whole.
- We instantiate generic classes using concrete types.

```
private class LinkedList<T>{
    public T head(){...}
    public void insert(T newdata){...}
}

LinkedList<Ticket> ticketList = new LinkedList<Ticket>();
```

- Be careful not to accidentally hide a type variable

```
public <T> void insert(T newDate){...}
```

- T in the argument of *insert*() is a new T, different from the classes T.
- Java array typing is covariant, If S extends T then $S[]$ extends $T[]$.
- Generic classes are not covariant, $LinkedList < String >$ is not compatible with $LinkedList < Object >$.
- **Wildcards**: ? stands for an arbitrary unknown type. Avoids unnecessary type variable quantification when the type variable is not needed elsewhere.

```
public static void printList(LinkedList<?> l){
    Object o;
    Iterator i = l.get_iterator();
    while(i.has_next()){
        o = i.get_next();
        System.out.println(o);
    }
}
```

- We can bound wild cards as follows

```
public static void drawAll(LinkedList<? extends Shape>){...}
```

- We can copy a linked list as follows

```
public static <? extends T, T> void listcopy(LinkedList<?> src, LinkedList<T>
public static <T, ? super T> void listcopy(LinkedList<T> src, LinkedList<?> t
```

3.14 Reflection

- **Introspection**: A program can observe, and therefore reason about its own state.
- **Intercession**: A program can modify its execution state or alter its own interpretation or meaning.
- Suppose we want to write a function to check if two different objects are both instances of the same class?

```
import java.lang.reflect;
class MyReflectionClass{
    ...
    public static boolean classequal(Object o1, Object o2){
        return o1.getClass() == o2.getClass();
    }
}
```

- *getClass*() returns an object of type *Class* that encodes class information.
- For each currently loaded class *C*, Java creates an object of type *Class* with information about *C*.
- We can create new instances of a class at runtime.

```
Class c = obj.getClass();
Object o = c.newInstance();
String s = "Manager"'
Class c = Class.forName(s);
Object o = c.newInstance();
```

- From the *Class* object we can extract more data as well

```
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
Method[] methods = c.getMethods();
Field[] fields = c.getFields();
```

- These in turn have functions to get further details.
- We can also invoke methods and examine/set values of fields

```
Class c = obj.getClass();
Constructor[] constructors = c.getConstructors();
Class params[] = constructors[0].getParameterTypes();
Method[] methods = c.getMethods;
Object[] args = {...};
methods[3].invoke(obj, args); // invoke methods[3] on obj with argument args
Field[] fields = c.getFields();
Object o = fields[2].get(obj);
fields[3].set(obj, value);
```

- All of these only extract publicly defined values.
- For private use *getDeclaredConstructors*(), similar for rest.
- **BlueJ**, a programming environment to learn Java.

3.15 Erasure of Generics

- **Type Erasure**: Java does not keep record of all versions of *LinkedList* $<T>$ as separate types. At run time, all type variables are promoted to *Object* or an upper bound if one is available.
- *LinkedList* $<T>$ becomes *LinkedList* $<$ *Object* $>$ and *LinkedList* $<?$ extends *Shape* $>$ becomes *LinkedList* $<$ *Shape* $>$.
- So, we cannot use if(o instanceof T)
- As a consequence *LinkedList* $<$ *Employee* $>$ and *LinkedList* $<$ *Date* $>$ are the same class.

- Wrapper classes for basic types

Basic type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long

Basic type	Wrapper Class
float	Float
double	Double
boolean	Boolean
char	Character

Figure 3: Wrapper Class

- All wrapper classes other than *Boolean*, *Character* extend the class *Number*.
- Converting from basic type to wrapper class and back, there is also autoboxing, i.e., implicit conversion

```
int x = 5;
Integer myx = Integer(x);
Integer myy = x;
int y = myx.intValue();
int x = myy;
```

3.16 Collections

- Most programming languages provide built-in collective data types, Arrays, Lists, Dictionaries.
- The *Collection* interface abstracts properties of grouped data, *Arrays*, *Lists*, *Sets*, any non key-value structures.
- Two methods *add*(), that adds to the collection. *iterator*(), get an object that implements *Iterator* interface.
- Iterator uses *hasNext*() and *next*() methods.
- Iterator also has *remove*() method, removes the element that was last accessed by *next*().
- The *Collection* interface implements a lot of methods

```
public interface Collection<E>{
    boolean add(E element);
    Iterator<E> iterator();
    int size();
    boolean isEmpty();
    boolean contains(Object obj);
    boolean containsAll(Collection<?> c);
    boolean equals(Object other);
    boolean addAll(Collection<? extends E> from);
    boolean remove(Object obj);
    boolean removeAll(Collection<?> c);
}
```

- *AbstractCollection* class provides default implementation of the interface.

3.17 Concrete Collections

- Interface *List* for ordered collections.
- *ListIterator* extends *Iterator* and adds *add(Eelement)*, *previous*(), *hasPrevious*() methods.

```
public interface List<E> extends Collection<E>{
    void add(int index, E element);
    void remove(int index);
    E get(int index);
    E set(int index, E element);
}
```

- *AbstractList* provides default implementation of *List*.
- Interface *Set* for collection without duplicates, identical to *Collection* but with constraints.

- *HashSet* implements a hash table. It is unordered, but supports *iterator*() that scans in unspecified order.
- *TreeSet* uses a tree representation, values are ordered and maintains a sorted collection.
- Interface *Queue* for ordered collections with constraints on addition and deletion.
- Can be *Dequeue* or *PriorityQueue*.

3.18 Maps

- Key value structures come under the *Map* interface.
- Two type parameters, K is for keys and V is for values.
- $get(k)$ fetches the value for the key k, and $put(k, v)$ updates the value for the key k.
- $put(k, v)$ returns the previous value associated with k.
- Also implements $getOrDefault(k, v')$
- Also implements $putIfAbsent(key, 0)$
- We also have *keySet*(), *valueSet*(), and *entrySet*().
- *HashMap* works similar to *HashSet*, there is no fixed order.
- *TreeMap* is similar to *TreeSet*, uses a balanced search tree to store. An iterator over *keySet*() will process the keys in a sorted order.
- *LinkedHashMap* remembers the order in which keys were inserted. Iterators will enumerate in the order of insertion. Similar to *LinkedHashSet*.

3.19 Error Handling

- Code could encounter many types of errors, it should kill itself.
- Code that generates an error raises or throws an exception.
- Caller catches the exception and takes corrective action or pass the exception back up the calling chain.
- All exceptions descend from class *Throwable* which has two branches *Error* and *Exception*.
- *Error* are relatively rare, "not the programmer's fault".
- *Exception* has two sub-branches, one of which is *RunTimeException*.
- Enclose the code that may generate an exception in a *try* block and the exception handler code should be in *catch* block.

```
try{
    call a function that may throw an exception
} catch(ExceptionType e){
    examine e and handle it
}
```

- Can catch more than one exception, catch blocks are tried sequentially.
- Order *catch* blocks by argument type, more specific to less specific.
- We can also *throw* a checked exception.
- Example: throw new EOFException(errormsg);

```
String readData(Scanner in) throws EOFException{
    ...
    while(...){
        if(n < len){
            throw new EOFException(errmsg);
        }
    }
}
```

- Can throw multiple types of exception. Can throw subtype of declared exception type.
- If we call such a method, we must handle it or pass it on.
- Can also create custom exception

```java
public class NegativeException extends Exception{
    private int error_value;
    // Negative value that generated exception
    public NegativeException(String message, int i){
        super(message); // Appeal to the super class
        error_value = i; // Constructor to set message
    }
    public int report_error_value(){
        return error_value;
    }
}
```

- We can extract information about the exception *e.getMessage*(), *getCause*(), *initCause*().
- When an exception occurs, rest of the *try* block is skipped. We may need to do some clean up, add a block labelled *finally*.

3.20 Packages

- Java has an organizational unit called *package*.
- Can use *import* to use packages directly, *import java.math.BigDecimal*
- Get all classes as *import java.math.*∗, ∗ is not recursive.
- We can declare packages as *package in.ac.iitm.onlinedegree*. Name is based on folder hierarchy, with *in* being the root folder.
- *protected* means visible within the subtree, so all subclasses. *protected* can be made *public*.

3.21 Assertions

- Functions may have constraints on the parameters, we "assert" the property we assume to hold.

```java
public static double myfn(double x){
    assert x >= 0 : x;
}
```

- If assertion fails, the code will throw *AssertionEror*, this should not be caught. Colon provides additional information to be printed with the diagnostic message.
- Assertions are enabled or disabled at runtime, does not require re-compilation.
- *java* −*enableassertions MyCode*, or we could use −*ea*.
- Can also selectively enable assertions as *java* −*ea* : *Myclass MyCode*
- Similar for disable assertions, −*da*.
- To enable assertions for system class, use −*esa* or enable system assertions.

3.22 Logging

- It is rather typical to generate messages within code for diagnosis. Naive approach is to use the print statements.
- Instead log diagnostic messages separately, logs are arranged hierarchically.
- Simplest info call

```java
Logger.getGlobal().info("Edit->Copy menu item selected");
```

- We can suppress logging by executing

```
Logging.getGlobal().setLevel(Level.OFF);
```

- Can also create custom logger

```
private static final Logger myLogger = Logger.getLogger("in.ac.iitm.onlinedegree");
```

- Logger names are hierarchical, like package names.
- There are seven logging level, SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST.
- By default, first three levels are logged
- We can set a different level

```
logger.setLevel(Level.FINE);
```

- Turn on or off all logging using Level.ALL and Level.OFF.

3.23 Cloning

- Normal assignment creates two references to the same object. Basically, two different variables point to the same object in memory.
- Object defines a method *clone*() that returns a bitwise copy.
- If the object that is being copied has an array then only the pointer is copied, so bitwise copy is **shallow copy**.
- For **deep copy** we can override the *clone*() method, first call *super* then replace all arrays with fresh copies.
- To allow *clone*(), class must implement *Cloneable* interface.
- *clone*() in Object is protected, redefine it to be public.
- *clone*() in Object throws CloneNotSupportedException which must be taken into account when overriding.

3.24 Streams

- Suppose we have split a text file as a list of words, we generate a stream of values from a collection.
- Operations transform input streams to output streams.

```
List<String> words = ...;
long count = words.stream().filter(w -> w.length() > 10).count();
```

- Stream processing is declarative, and processing can be parallelized parallelStream().
- Lazy evaluation is possible.
- A stream does not store its elements.
- Stream operations are non-destructive, input stream is untouched.
- We create a stream, transform it and then reduce it to a result.
- Apply *stream*() to a collection. Use static method *Stream.of*() for arrays.
- *Stram.generate*() generates a stream from a function and *Stream.iterate*() is a stream of dependent values.

```
Stream<String> randomds = Stram.generate(Math::random);
Stream<Integer> integers = Stream.iterate(0, n -> n < 100, n -> n + 1);
```

- *filter*() to select elements, takes a predicate as arguments.
- *map*() applies a function to each element in the stream.
- If *map*() function generates a list, we can instead use *flatMap*() to flatten the list.
- *limit*(n), makes a stream finite.
- *skip*(n), skips first n elements.

- $takeWhile()$, stop when element matches a criterion.

```
Stream<Double> randomds = Stream.generate(Math::random).takeWhile(n -> n >= 0
```

- $dropWhile()$, start after element matches a criterion.
- $count()$, count the number of elements.
- $max()$ and $min()$, requires a comparison function.
- $findFirst()$, returns the first element.
- If the stream is empty then the termination functions will return null or no object.
- We can use $orElse()$, $orElseGet()$, $orElseThrow()$

```
Double fixrand = maxrand.orElse(-1.0);
```

- We can also ignore missing values $ifPresent(v- > Processv)$, or $isPresentOrElse$
- We can create an optional value, $Optional.of(v)$ or $Optional.empty()$. Can also use $Optional.ofNullable()$ to transform $null$ into an empty optional.
- We can produce an output $Optional$ value from an input $Optional$.
- Can convert a stream into an array using $toArray()$.
- Can use $collect(Collectors.toList())$ to convert stream back into a collection. It can be any collection, doesn't have to be list.
- $summarizingInt$ works for a stream of integers that stores count, max, min, sum, average. Can be accessed using $getCount()$.
- We can convert stream to map as well, similar syntax, requires what key and value to take. Can store entire object as value using $Function.identity()$.
- We can read and write raw bytes using $InputStream$ and $OutputStream$.
- Use $in.available()$ to check if input is available, and $in.read(data)$ to read data and $in.close()$ to close the stream.
- Use $out.write(values)$ to write values to the output, and $out.flush()$ to flush the output, output is buffered.
- We can also connect to file using $FileInputStream(file)$ and $FileOutputStream(file, boolean)$. Boolean decides whether we are overwriting(false) or appending(true).
- Use $Scanner$ class as new Scanner(fin), where fin is $FileInputStream$.
- To write text use $PrintWriter$ class, has methods $println$ and $print$.
- To write binary data, use $DataOutputStream$ class.
- Fucking java has a zoo of streams.

4 Philosophy of OOPs

4.1 Introduction

- Traditionally, algorithm comes first, and data representation comes later.
- OOPs reverses this focus.
- We design objects with few key points in mind.
- **Behavior**: What methods do we need to operate on objects?
- **State**: How does the object react when methods are invoked?
- **Encapsulation**: Should not change unless a method operates on it.
- **Identity**: Distinguish between different objects of the same class. State may be the same.
- Robust design minimizes dependencies, or coupling between classes.

4.2 Subclasses and Inheritance

- A typical Java class

```
public class Employee{
    private String name;
    private double salary;

    /* Constructors... */

    // Mutator methods
    public boolean setName(String s){...}
    public boolean set Salary(double x){...}

    // Accessor methods
    public String getName(){...}
    public int getSalary(){...}

    // Other methods
    public double bonus(float percent){
        return (percent/100.0) * salary'
    }
}
```

- Managers are special types of employees with extra features.
- *Manager* object inherits other fields and methods from *Employee*. *Manager* is a *subclass* of *Employee*.
- *Manager* objects do not automatically have access to private data of the parent class.
- Use parent class's constructor using *super*.

```
public class Manager extends Employee{
    ...
    public Manager(String n, double s, String sn){
        super(n,s);
        secretary = sn;
    }
}
```

- In general, subclass has more features than parent class.
- Every *Manager* is an *Employee*, but every *Employee* is not a *Manager*.

```
Employee e = new Manager(...);
```

- The above works, but reverse does not.

4.3 Dynamic dispatch and Polymorphism

- *Manager* can redefine *bonus*(). Use parent class *bonus* via *super.bonus*().
- Consider the following assignment

```
Employee e = new Manager(...);
```

- *e* can only refer to methods in *Employee*.
- If a method is defined in both *Employee* and *Manager*, then *e* will consider method defined in *Manager*, this is **dynamic dispatch**.
- Signature of a function is its name and the list of argument types.
- Can have different functions with the same name and different signatures.
- Java class *Arrays* has a method *sort* to sort arbitrary scalar arrays.

- **Overloading**: multiple methods, same name, different signature(different parameters), choice is static.
- **Overriding**: multiple methods, same name, same signature, choice is static.
- **Dynamic Dispatch**: multiple methods, same signature, choice made at run-time.
- Consider the method *getSecretary*(), which is only defined in *Manager* class.
- The earlier definition of *e* cannot invoke this method directly, instead we need **type casting**.

```
((Manager) e).getSecretary();
```

- Cast fails at run time if *e* is not a *Manager*.
- We can test if *e* is *Manager* as

```
if(e instanceof Manager){
    ((Manager) e).getSecretary();
}
```

- We can also use type casting for basic data types.

4.4 Benefits of Indirection

- To use different implementation of *Queue* or any other data structure.

```
Queue<Date> dateq;
Queue<String> stringq;
dateq = new CircularArrayQueue<Date>();
stringq = new LinkedListQueue<String>();
```

5 Concurrent Programming

5.1 Threads and Processes

- Multiprocessing, single processor executes several computations "in parallel".
- It's basically switching between different actions very fast.
- Have a class extend *Thread* and define a function *run*() where execution can begin in parallel.
- Invoking *start* on the object will start *run* in a separate thread.

```
public class Parallel extends Thread{
    private int id;
    public Parallel(int i){id = i;}
    public void run(){
        for(int j = 0; j < 100; j++){
            System.out.println("id: " + id);
        }
        try{
            // Sleep for 1000ms
            sleep(1000);
        }catch(InterruptedException e){...}
    }
}
```

- Directly calling *run* will also execute in a separate thread.
- Since, we cannot always extend *Thread* we instead implement *Runnable*.
- To use *Runnable*, we need to explicitly create a *Thread* object and invoke *start* on the object.

5.2 Race Conditions

- **Race Condition**: Concurrent update of shared variables, unpredictable outcome.
- Mutually exclusive access to critical regions of code, make sure two threads don't access the shared variable at once.
- We can try to do this by introducing another shared variable *turn*, that decides which thread has access. The problem with this, if one thread shuts down randomly, then others are locked out permanently leading to **Starvation**.
- Another approach can be as follows, but this can lead to **Deadlock**, where both threads try simultaneously.

```
Thread 1
...
request_1 = true;
while (request_2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
while (request_1)
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

(a) Deadlock

```
Thread 1
...
request_1 = true;
turn = 2;
while (request_2 &&
         turn != 1){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_1 = false;
...
```

```
Thread 2
...
request_2 = true;
turn = 1;
while (request_1 &&
         turn != 2){
  // "Busy" wait
}
// Enter critical section
   ...
// Leave critical section
request_2 = false;
...
```

(b) Peterson's Algorithm

- We can combine the previous two approaches into **Peterson's algorithm**.
- Generalizing this to more than two processes is not trivial.
- **Lamport's Bakery Algorithm**: Each new process picks p a token(increments a counter) that is larger than all waiting processes, the lowest token number gets served next, we still need to break ties.

5.3 Semaphores

- The fundamental issue preventing consistent concurrent updates of shared variables is $test - and - set$.
- To increment a counter, check its current value, then add 1. If more than one thread does this in parallel, updates may overlap and get lost.
- Need to combine test and set into an atomic, indivisible step.
- Semaphores are a programming language's support for mutual exclusion.
- A semaphore s supports two atomic operations
- $P(s)$: from Dutch *passeren*, to pass.
- $V(s)$: from Dutch *vrygeven*, to release.

```
if (S > 0)
  decrement S;
else
  wait for S to become positive;
```

(a) P(s) automatically executes this

```
if (there are threads waiting
     for S to become positive)
  wake one of them up;
     //choice is nondeterministic
else
  increment S;
```

(b) V(s) automatically executes this

- Semaphores guarantee mutual exclusion, freedom from starvation and deadlock.
- However, they are too low level, and there is no clear relationship between a semaphore and the critical region that it protects.

5.4 Monitors

- Attach synchronization control to the data that is being protected.
- Monitors are like a class, variables are the data that needs to be protected and functions are the ones that modify this data.
- Monitor guarantees mutual exclusion, if one function is active, any other function will have to wait for it to finish.
- *wait*(), all other processes are blocked out while this process waits.
- *notify*(), signals and exits.

```
monitor bank_account{
    double accounts[100];
    queue q[100]; // one internal queue for each account
    boolean transfer(double amount, int source, int target){
        while(accounts[source] < amount){
            q[source].wait(); // wait in the queue associated with source
        }
        accounts[source] -= amount;
        accounts[target] += amount;
        q[target].notify(); // notify the queue associated with target
        return true;
    }
}
```

- In java, this is written as

```java
public class bank_account{
    double accounts[100];
    public synchronized boolean transfer(double amount, int source, int targe
        while(accounts[source] < amount){wait();}
        accounts[source] -= amount;
        accounts[target] += amount;
        notifyAll();
        return true;
    }
    public synchronized double audit(){
        double balance = 0.0;
        for(int i = 0; i < 100; i++){
            balance += accounts[i]
        }
        return balance;
    }
    public double current_balance(int i){
        return accounts[i]; // not synchronized
    }
}
```

- Rest of this is much easier to learn by practicing, there is no point in having notes for code.

6 Graphical User Interfaces

This is just easier to look up tutorials or projects to understand. Use the Swing Toolkit. Not that hard to use, checkout my game I made using it, it has no sound and is very buggy, but it works.