



# সূচিপত্র

পরিচিতি	0
উপক্রমণিকা	1
পাঠ ১: তোমার প্রথম জাভা প্রোগ্রাম	2
পাঠ ২: সিনট্যাক্স	3
পাঠ ৩: ডাটা টাইপস এবং অপারেটর	4
পাঠ ৩.১: এরে	4.1
পাঠ ৩.২: এক্সপ্রেশন(Expressions), স্টেটমেন্ট (Statements) এবং ব্লক(Blocks)	4.2
পাঠ ৪: কন্ট্রোল ফ্লো -লুপিং- ব্রাঞ্চিং	5
পাঠ ৫: অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং-১	6
পাঠ ৫.১: ইনহেরিট্যান্স	6.1
পাঠ ৫.২: পলিমরফিজম	6.2
পাঠ ৫.৩: এনক্যাপসুলেশন	6.3
পাঠ ৬: জাভা এক্সপ্রেশন হ্যান্ডেলিং	7
পাঠ ৬.১: ক্লোজার লুক	8
পাঠ ৭: স্ট্রিং অপারেশন	9
পাঠ ৮: জেনেরিকস	10
পাঠ ৯: জাভা আই/ও	11
পাঠ ১০: জাভা এন আই/ও	12
পাঠ ১১: জাভা কালেকশন ফ্রেমওয়ার্ক	13
পাঠ ১২: জাভা জেডিবিসি	14
পাঠ ১৩: জাভা লগিং	15
পাঠ ১৪: ডিবাগিং	16
পাঠ ১৫: গ্রাফিক্যাল ইউজার ইন্টারফেইস	17
পাঠ-১৬: থ্রেড	18
পাঠ ১৭: নেটওয়ার্কিং	19
পাঠ ১৮: জাভা কনকারেন্সি	20
পাঠ ১৯: ক্লাস ফাইল এবং বাইটকোড	21
পাঠ ২০: Understanding performance tuning	22
পাঠ ২১: মডার্ন জাভা ইউজেস	23
অনুশীলন	24



# জাভা প্রোগ্রামিং

 Like  Share 9.9K people like this. [Sign Up](#) to see what your friends like.

স্বয়ংক্রিয় কন্ট্রিবিউটরের তালিকা  
(প্রথম ৫ জন)

[\[139\] A. N. M. Bazlur Rahman \(Rokon\)](#)  
[\[012\] Nuhil Mehdy](#)  
[\[012\] Md. Ashikuzzaman](#)  
[\[004\] Nazmul Hussain](#)  
[\[002\] Mushrit Shabnam](#)

## সংক্ষেপ

কোর্সের বর্ণনা: জাভা বর্তমানে বহুল ব্যবহৃত একটি প্রোগ্রামিং ল্যাংগুয়েজ। এন্টারপ্রাইজ এপ্লিকেশান ডেভেলেপমেন্টে এখনো জাভার বিকল্প তৈরি হয়নি বলে ধরা হয়। জাভার জনপ্রিয়তার মূল কারণ এর portability, নিরাপত্তা, এবং অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং ও ওয়েব প্রোগ্রামিং এর পরিপূর্ণ সাপোর্ট। এই কোর্সে জাভার অ আ ক থ থেকে শুরু করে এর ব্যবহারিক প্রয়োগ এবং অন্যান্য বিষয় গুলো নিয়ে আলোচনা করা হবে।

কাদের জন্যে কোর্স: এই কোর্স মূলত বিশ্ববিদ্যালয় এর প্রথম বর্ষের ছাত্র-ছাত্রীদের জন্যে যারা অবজেক্ট ওরিয়েন্টেড কনসেপ্ট শুরু করতে চায়। তবে যে কেও চাইলে এই কোর্সটি করতে পারে। ধরে নেওয়া হচ্ছে যে, শিক্ষার্থী অন্ত্যত যে কোন একটি প্রোগ্রামিং ল্যাংগুয়েজ (সি/সি++) সম্পর্কে আগে থেকেই ধারণা রাখে।




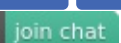
## Statutory warning

This book may contain unexpected misspellings. Reader Feedback Requested.

## ওপেন সোর্স

এই বইটি মূলত স্বেচ্ছাশ্রমে লেখা এবং বইটি সম্পূর্ণ ওপেন সোর্স। এখানে তাই আপনিও অবদান রাখতে পারেন লেখক হিসেবে। আপনার কন্ট্রিবিউশান গৃহীত হলে অবদানকারীদের তালিকায় আপনার নাম যোগ করে দেওয়া হবে।

এটি মূলত একটি [গিটহাব রিপোজিটরি](#) যেখানে এই বইয়ের আর্টিকেল গুলো মার্কডাউন ফরম্যাটে লেখা হচ্ছে। রিপোজিটরিটি ফর্ক করে পুল রিকুয়েস্ট পাঠানোর মাধ্যমে আপনারাও অবদান রাখতে পারেন। বিস্তারিত দেখতে পারেন এই ভিডিওতে [Video](#)

 Like 170  Share  
 gitter  join chat



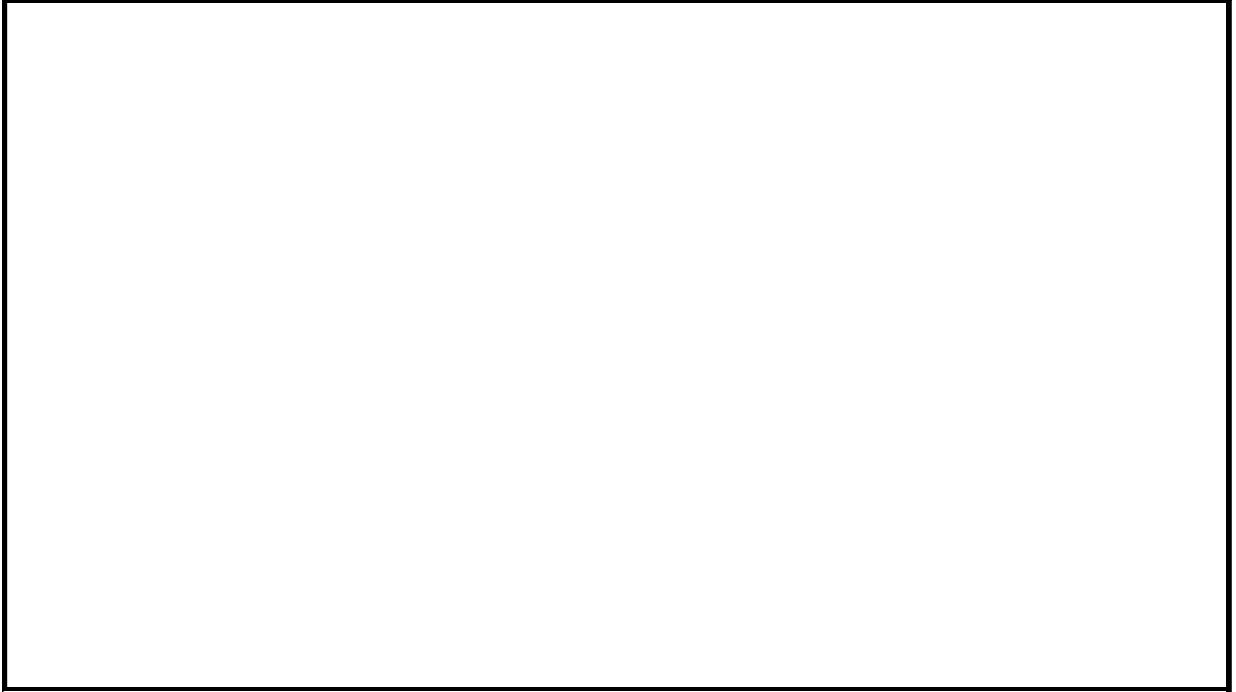
This work is licensed under a [Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License](https://creativecommons.org/licenses/by-nc-nd/4.0/).

## উপক্রমণিকা

১৯৯৫ সালের ২৩ শে মে। ঝকঝকে ঝলমলে চমৎকার একটি দিন। জন গেইজ, ডিরেক্টর অব সান মাইক্রোসিস্টেম সাথে Marc Andreessen, কো ফাউন্ডার এবং ভাইস প্রেসিডেন্ট অব নেটস্কেপ ঘোষণা দেন যে, জাভা টেকনোলজি মোটেই কোন উপকথা নয়, বরং এটিই বাস্তবতা এবং তারা এটি Netscape Navigator এর সংযুক্ত হতে যাচ্ছে।

সে সময় জাভাতে কাজ করে এমন লোকের সংখ্যা ত্রিশেরও কম। তারা কখনোই চিন্তা করে নি, তাদের এই টিম ভবিষ্যৎ পৃথিবীর প্রধানতম টেকনোলজি নির্ধারণ করতে যাচ্ছে। ২০০৪ সালের ৩ জানুয়ারী Mars Exploration Rover মঙ্গল গ্রহের মানটিতে পা রাখে যার কন্ট্রোল সিস্টেম থেকে শুরু করে পৃথিবীর অধিকাংশ কনজুমার ইলেকট্রনিক্স - (ক্যাবল সেট-টব বক্স, ডিসিআর, টোস্টার, পিডিএ, স্মার্টফোন) ৯৭% এন্ট্রাপ্রাইজ ডেস্কটপ ৮৯% ডেস্কটপ অব ইউএসএ, ৩ বিলিওন মোবাইল ফোন, ৫ বিলিওন জাভা কার্ড, ১২৫ মিলিওন টিভি ডিভাইস, ১০০% ব্লু-রে ডিস্ক প্লেয়ার ... এই লিস্ট লম্বা হতেই থাকবে) জাভা রান করে।

নিচের ভিডিও টি চমৎকার। একবার দেখে নেওয়া যেতে পারে।



চলুন একটু পেছনের ইতিহাস জেনে নেই।

তখন সি-প্লাস প্লাস এর একচ্ছত্রাধিপত্য।

সান মাইক্রোসিস্টেম- মূলত হার্ডওয়্যার কম্পানি। ১৯৭২ থেকে ১৯৯১ সালে কম্পিউটারের হার্ডওয়্যারের এক বেডু্যলেশান হয়। দ্রুত এবং উচ্চ ক্ষমতা সম্পন্ন হার্ডওয়্যার অল্প দামে পাওয়া যাচ্ছে এবং সেই সাথে কমপ্লেক্স সফটওয়্যারের চাহিদা দ্রুতই বেড়ে যাচ্ছে। ১৯৭২ Dennis Ritchie সি প্রোগ্রামিং ল্যাংগুয়েজ ডেভেলপ করেন যা প্রোগ্রামারদের মধ্যে সব থেকে জনপ্রিয়। কিন্তু ততদিনে প্রোগ্রামারদের কাছে সি -এর স্ট্রাকচার্ড প্রোগ্রামিং কিছুটা ক্লান্তিকর মনে হতে শুরু করেছে। এর ফলশ্রুতিতে Bjarne Stroustrup 1979 সালে ডেভেলপ করে সি প্লাস প্লাস যা কিনা সি এর এনহান্সমেন্ট। এটি সাথে অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং ধারণাকে পরিচিত করে তুলে। অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর সুবিধে হচ্ছে প্রোগ্রামার পুনর্ব্যবহারযোগ্য(reusable) কোড লিখতে পারে যা কিনা পরে অন্য কাজে পুনরায় ব্যবহার করা যায়।

১৯৯০ সাল। সান মাইক্রোসিস্টেম -এ সি প্লাস প্লাস এর আধিপত্যে সি-তে লেখা টুল এবং এপিআই গুলো প্রায় অবস্যুলেট হতে শুরু করেছে। Patrick Naughton, ইঞ্জিনিয়ার অব সান মাইক্রোসিস্টেম, মোটামুটি হতাশ এবং এক ধরনের অকওয়াড পরিস্থিতির স্বীকার। ততদিনে স্টিভ জব অ্যাপল কম্পিউটার থেকে বিতাড়িত হয়ে NeXT Computer, Inc প্রতিষ্ঠা করে ফেলেছেন( যা কিনা সফটওয়্যার ইন্ডাস্ট্রিতে বৈপ্লবিক পরিবর্তন আনতে যাচ্ছে এবং যার ফলশ্রুতিতে তৈরি হয়েছে আজকের ম্যাক-ওস) এবং NeXTSTEP নামে একটি অপারেটিং সিস্টেম তৈরি করেন। এতে কিছু অসাধারণ ব্যাপার ছিল যার মধ্যে অবজেক্ট ওরিয়েন্টেড এপ্লিকেশান লেয়ার এর ধারণাটি ছিল অসাধারণ যাতে কিনা অবজেক্ট ধরে ধরে কাস্টমাইজড সফটওয়্যার তৈরি করে ফেলা যায়। Patrick Naughton ইতিমধ্যে NeXT এর দিকে যাওয়ার জন্য মনস্থির করে ফেলেছেন কিন্তু তখন একবার তাকে শেষ সুযোগ হিসেবে একটি অতি গোপন প্রজেক্টের অনুমোদন দেওয়া হয় যার কথা কেউ জানতো না। কিছুদিন পরেই তার সাথে যুক্ত হয় James Gosling এবং Mike Sheridan। তখন এর নাম দেওয়া হয় গ্রিন প্রজেক্ট। সময়ের সাথে গ্রিন প্রজেক্ট এর দস্তোদাম হয় এবং তারা কম্পিউটার ছাড়াও বিভিন্ন ডিভাইস নিয়ে নার্চার করতে থাকে।

এর মধ্যে ১৩ জন স্টাফ এই গ্রিন টিম ক্যালিফোর্নিয়ার মেনলো পার্কের সেন্ড হিল রোড এর একটি ছোট্ট অফিসে কাজ করতে থাকে। তাদের প্রধান উদ্দেশ্য সি প্লাস প্লাস এর একটি ভাল ভার্সন তৈরি করা যা কিনা হবে অনেক দ্রুতগামী এবং রেস্পন্সিভ। সেই সময়ে কম্পিউটার ছাড়াও কনজুমার ইলেকট্রনিক্স যেমন -পিডিএ, Cable-Set Top Box ইত্যাদির চাহিদা বেড়ে গেছে। একদল ইঞ্জিনিয়ার এক সাথে থাকলে যা হয়, তারা নানারকম জিনিস নিয়ে চিন্তা করতে থাকে, নানা রকম আইডিয়া তৈরি হয়, তা থেকে প্রোটোটাইপ তৈরি করতে থাকে। এর মধ্যে জেমস গসলিং তার সি প্লাস প্লাস এনহান্সমেন্ট চালিয়ে যেতে থাকেন। তিনি এর নাম দেন সি প্লাস প্লাস প্লাস প্লাস মাইনাস মাইনাস (C++ ++ -- )। এখানে বাড়তি ++ মানে হচ্ছে নতুন জিনিস যোগ করা এবং -- মানে হচ্ছে কিছু জিনিস ফেলে দেওয়া। জেমস গসলিং এর জানালা দিয়ে একটি ওক গাছ দেখা যায়। একদিন তিনি অফিস থেকে বের হয়ে ঐ গাছটির নিচে দাড়ান এবং সাথে সাথে C++ ++ -- নাম পরিবর্তন করার সিদ্ধান্ত নেন এবং নতুন নাম দেন ওক।

এর মধ্যে ইঞ্জিনিয়াররা মিলে এমবেডেড সিস্টেম নিয়ে নার্চার করতে থাকা অবস্থায় নানা রকম সমস্যার সন্মুখীন হন। এমবেডেড সিস্টেম এ মেমরি কম থাকে, প্রসেসিং পাওয়ার ও কম থাকে। এই সিস্টেমে সি++ (যা কিনা কম্পিউটার এর মতো বড় ফ্লুটিপ্রিস্টের হার্ডওয়্যারের জন্যে ডিজাইন করা) চালাতে গিয়ে তারা অদ্ভুত অদ্ভুত সমস্যার সন্মুখীন হতে থাকে। এইসব সমস্যার সমাধান করার জন্যে গ্রিন টিম নানা রকম চিন্তা ভাবনা করতে থাকে। এই সময়ে মানুষ পিডিএ, Cable-Set Top Box গুলোর মরণদশা দেখতে শুরু করে। কারণ যদিও ওক নিয়ে যথেষ্ট এগিয়েছে কিন্তু এটি কোনভাবেই এদেরকে সাহায্য করতে পারছিল না। একমাত্র একটি অলৌকিক ঘটনায় পারে এই প্রজেক্ট সফল করতে। ঠিক তখনই সেই প্রতীক্ষিত প্রত্যাশা আলোর মুখ দেখে। জেমস জেমস গসলিং আউট অব দ্যা বক্স একটা যুগান্তকারী ধারণা নিয়ে আসে। সেটি হলো ভার্সুয়াল মেশিন। অর্থাৎ আমরা একটা কাল্পনিক মেশিনের জন্যে কোড লিখবো যা কিনা কম্পাইল হয়ে একটি অন্তর্বর্তীকালীন কোড তৈরি করবে। এবং জাভা ভার্সুয়াল মেশিন সেই অন্তর্বর্তীকালীন কোডকে রান টাইম-এ রিয়েল ডিভাইসের জন্যে প্রয়োজন অনুযায়ী মেশিন কোড তৈরি করবে।

ঠিক সেই সময়েই National Center for Supercomputing Applications (NCSA) একটি কমার্শিয়াল ওয়েব ব্রাউজার বের করে এবং তাদের টিম ইন্টারনেট এর ভবিষ্যৎ নিয়ে ভাবতে শুরু করে। তারা একটি নতুন ধারণা নিয়ে আসে সেটি হলো, একধরনের ছোট্ট প্রোগ্রাম যা কিনা ব্রাউজার এর মধ্যে চলবে - এর নাম দেয় অ্যাপলেট। অ্যাপলেট ধারণা থেকে তারা ঠিক করে অ্যাপলেট এর জন্যে কিছু স্ট্যান্ডার্ড – এটি হতে হবে ছোট্ট, খুব সিম্পল, এর স্ট্যান্ডার্ড এপিআই থাকতে হবে, এটি হবে প্লাটফর্ম ইন্ডিপেন্ডেন্ট, এবং আউট-অব-দ্যা বক্স নেটওয়ার্কিং প্রোগ্রামিং করা যাবে। তারা তখনকার সময়ের ইন্টারনেট বুমকে উদ্দেশ্য করে নেট্রট জেনারেশান প্রোডাক্ট ডেভেলপ করতে চেয়েছিল। এই প্রজেক্ট এর কার্টুন নাম ছিল Duke ( যা কিনা এখন জাভা-এর মাস্কট হিসেবে চিনি)। কিন্তু সমস্যা হচ্ছে এর কোনটিই ঠিক মতো সি++ দিয়ে করা যাচ্ছিল না। সুতরাং পরবর্তীতে তারা সিদ্ধান্ত নেয় যে এমবেডেড সিস্টেমের সমস্যার সমাধানটি তারা ওয়েব ব্রাউজার এর ক্ষেত্রেও ব্যবহার করবে। সেই সময়ে মানুষ ওয়েব ব্রাউজার এর শুধুমাত্র স্ট্যাটিক

পেইজ এ টেক্সট আর ইমেজ ছাড়া কিছু দেখতে পেরে না। এই টেকনোলজি ব্যবহার করায় ব্রাউজার এনিমেশন থেকে শুরু করে ইন্টারেক্টিভ অ্যাপলেট সকলের নজর কাড়ে যা কিনা জাভা প্রোগ্রামিং ল্যাংগুয়েজ এর সফলতার মূল কারণ।

জেমস গসলিং এর এই ভার্সুয়াল মেশিন-এর সল্যুশন ছিল সত্যিকার অর্থেই যুগান্তকারী এবং স্মিথ টিম এর রিলিজ দিতে প্রস্তুত। কিন্তু তখন-ই নতুন ঝামেলার সূচনা হয়, lawyers এসে তাদের জানায় এর নাম Oak দেওয়া যাবে না, কারণ এটি ইতিমধ্যেই Oak Technologies এর ট্রেড মার্ক। সুতরাং নাম পরিবর্তন করতে হবে। শুরু হয় ব্রেইনস্টর্মিং। কিন্তু কোন ভাবেই একটি ভাল নাম নির্বাচন করা যাচ্ছিল না। অনেকেই অনেক ধরনের নাম উপস্থাপন করে, যেমন - DNA, Silk, Ruby, yuck, Silk, Lyric, Pepper, NetProse, Neon, Java ইত্যাদি ইত্যাদি। এর সব গুলো লিগাল ডিপার্টমেন্ট এ সাবমিট করার পর মাত্র Java, DNA, and Silk এই তিনটি নাম ফিরে আসে যা কিনা স্ক্রিন। নাম নিয়ে ঘণ্টার পর ঘণ্টার মিটিং চলতে থাকে। এর মধ্যে Chris Warth প্রপোজ করে Java, কারণ তখন তার হাতে ছিল এক কাপ গরম Peet's Java (কফি)। শেষ পর্যন্ত নাম ঠিক করা হয় Java কারণ একমাত্র এই নামেই সব থেকে পজিটিভ রিএকশন পাওয়া যাচ্ছিল।

১৯৯৫ সালের মে মাসে জাভা এর প্রথম পাবলিক ভার্সন রিলিজ হয়।

এর পরের ইতিহাস আমরা সবাই জানি। জাভা হচ্ছে এই গ্রহের সবচেয়ে সফল প্রোগ্রামিং ভাষা।

## তোমার প্রথম জাভা প্রোগ্রাম

আমরা এই চ্যাপ্টার এ যে যে বিষয়গুলো দেখবো সেগুলো হলো-

- প্রোগ্রামিং ল্যাংগুয়েজ কি এবং কেন
- কেন জাভা
- জাভা কিভাবে কাজ করে, ভেতরের বৃত্তান্ত
- জাভা একটি কম্পাইল্ড ল্যাংগুয়েজ না ইন্টারপ্রেটেড ল্যাংগুয়েজ
- জাভা ভার্সুয়াল মেশিন কি এবং কিভাবে কাজ করে
- জাভা রানটাইম
- জাভা ডেভেলপমেন্ট কিট এবং আইডিই
- জেডিকে ইনস্টলেশন
- একটি হ্যালো ওয়ার্ল্ড প্রোগ্রাম

প্রোগ্রামিং ল্যাংগুয়েজ কি ?

প্রোগ্রামিং ল্যাংগুয়েজ হচ্ছে এক ধরনের কৃত্রিম ভাষা যা কিনা যন্ত্র বিশেষ করে কম্পিউটার-এর আচরণ নিয়ন্ত্রণ করার জন্যে ব্যবহার করা হয়। মানুষের ভাষার মতো এর কিছু সিনট্যাক্স এবং সেম্যান্টিক্স অর্থাৎ নিয়মকানুন ও অর্থ থাকে। আমাদের এই বই এর উদ্দেশ্য হচ্ছে একটি বিশেষ ভাষার(জাভা) নিয়মকানুন গুলো জেনে নেওয়া। সুতরাং পড়তে থাকুন।

কেন জাভা?

পৃথিবীতে এখন পর্যন্ত অনেক গুলো প্রোগ্রামিং ভাষা তৈরি করা হয়েছে। এদের প্রত্যেকটির উদ্দেশ্য ভিন্ন ভিন্ন।

[http://en.wikipedia.org/wiki/List\\_of\\_programming\\_languages](http://en.wikipedia.org/wiki/List_of_programming_languages) এখানে একটি প্রোগ্রামিং ল্যাংগুয়েজ এর একটি লিস্ট দেওয়া আছে- দেখে নেওয়া যেতে পারে। প্রত্যেকটি ল্যাংগুয়েজ এর কিছু সুবিধা অসুবিধা আছে, এবং ল্যাংগুয়েজ গুলো প্রতিনিয়ত উন্নত হচ্ছে, এবং নতুন নতুন ল্যাংগুয়েজ তৈরি হচ্ছে।

যে যে কারণে জাভা শেখা যেতে পারে এখন সেগুলো নিয়ে আলোচনা করা যাক-

- এটি খুব-ই (Readable) পাঠযোগ্য, সহজে বুঝা যায়। অন্য যে কোন প্রোগ্রামিং ব্যাকগ্রাউন্ড এর প্রোগ্রামার খুব সহজেই একটি জাভা-ফাইল দেখে বুঝতে পারবে আসলে কোড এ কি লেখা আছে।
- সি কিংবা সি++ এ কোড করার সময় আমাদের অনেক সময়-ই লিংকিং, অপটিমাইজেশান, মেমরি এলোকেশান, মেমরি ডি-এলোকেশান, পয়েন্টার ডিরেফারেন্সিং ইত্যাদি নানা রকম জিনিস নিয়ে ভাবতে হয়, কিন্তু জাভার ক্ষেত্রে এগুলোর কথা ভাবতেই হয় না। খুব বেশি চিন্তা না করে আমরা নিশ্চিতভাবে জাভা কম্পাইলার এর উপর সব কিছু ছেড়ে দিতে পারে।
- জাভাতে অসংখ্য API আছে যেগুলো খুবই স্টেবল, খুব বেশি চিন্তাভাবনা না করেই এদের নিয়ে খুব সহজেই কাজ করে ফেলা যায়।
- জাভা -র সব কিছুই ওপেন সোর্স।
- জাভা ভার্সুয়াল মেশিন সম্ভবত সফটওয়্যার- জগতে সব থেকে চমৎকার সৃষ্টি। জাভা-এর সাথে এর আরও অনেকগুলো ল্যাংগুয়েজ যেমন- গ্রুভি, স্ক্যালা ইত্যাদি নিয়ে কাজ করা যায়।



- গত ১৫ বছরে চমৎকার অনেকগুলো ডেভেলপমেন্ট এনভায়রনমেন্ট তৈরি হয়েছে যেগুলো খুবই ইন্টেলিজেন্ট – যেমন- Eclipse, IntelliJ IDEA, netbeans etc. । এগুলো মাধ্যমে খুব আয়শের সাথেই কোড করা যায়, ডিবাগ করা যায় ।
- এটি একটি অবজেক্ট ওরিয়েন্টেড- টাইপ সেইফ প্রোগ্রামিং ল্যাংগুয়েজ ।
- এটি পোর্টেবল যে কোন প্ল্যাটফর্মে চলে । একবার কোড লিখে সেটি যে কোন মেশিনে( উইন্ডোজ , লিনাক্স , ম্যাক ) চালানো যায় ।
- অনেক বড় কমিউনিটি সাপোর্ট- সারা দুনিয়াতে মিলিয়নস অব জাভা প্রোগ্রামার ছড়িয়ে ছিটিয়ে আছে ।
- এটির পারফরমেন্স নিয়ে বলা চলে কোন সন্দেহ নেই ।
- ইন্ডাস্ট্রি গ্রেডেড, বড় বড় এন্টারপ্রাইজ অ্যাপ গুলো সাধারণত জাভা দিয়ে লেখা হয় ।
- এটি পৃথিবীতে দ্বিতীয় জনপ্রিয় ল্যাংগুয়েজ-

<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

এই লিস্ট এখানেই থামিয়ে দেই- কারণ এটি শেষ হতে চাইবে না কখনোই ।

জাভা কিভাবে কাজ করে ?

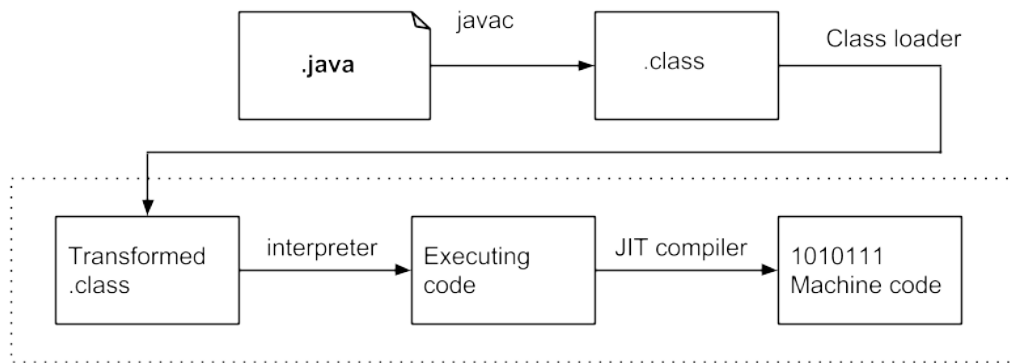


Figure: How java works

জাভা কোডকে কম্পাইল করলে সেটি একটি অন্তর্বর্তীকালীন ল্যাংগুয়েজ এ রূপান্তরিত হয় । এটি ঠিক হিউম্যান রিডএবল না আবার মেশিন রিডএবল ও না । একে আমরা বলি বাইট কোড । এই বাইটকোড শুধুমাত্র জাভা ভার্সুয়াল মেশিন(JVM) বুঝতে পারে । JVM বাইট কোড কে ইন্টারপ্রেট করে মেশিন ল্যাংগুয়েজ এ রূপান্তরিত করে । এর জন্যে JVM জাস্ট ইন টাইম(JIT) কম্পাইলার ব্যবহার করে । সুতরাং দেখা যাচ্ছে, জাভা কোডকে প্রথমে কম্পাইল করা হয়, তারপর সেই আউটপুট কে ইন্টারপ্রেট করা হয় । এক্ষেত্রে প্রশ্ন হতে পারে, জাভা আসলে কি? কম্পাইল্ড ল্যাংগুয়েজ নাকি ইন্টারপ্রেটেড ল্যাংগুয়েজ? উত্তর হচ্ছে জাভা একি সাথে দুটোই ।

উপরের বর্ণনা থেকে আমরা তিনটি জিনিস জানলাম -

১. বাইট কোড – এটি হচ্ছে এক ধরনের ইন্সট্রাকশান সেট- যা কিনা শুধুমাত্র জাভা ভার্সুয়াল মেশিন বুঝতে পারে । জাভা কোড ( হিউম্যান রিডএবল ) অর্থাৎ আমরা যে কোড গুলো লিখবো সেগুলো কে জাভা কম্পাইলার দ্বারা কম্পাইল করলে বাইটকোড তৈরি হয় । এই বাইটকোড গুলো .class এক্সটেনশন যুক্ত বাইনারী ফাইলে স্টোর করা হয় ।

২. জাভা ভার্চুয়াল মেশিন(**JVM**) - এটি মূলত একটা বাস্তব মেশিনের ভেতর একটা কাল্পনিক মেশিন। সহজ কথায়- এটি একটি সফটওয়্যার যা কিনা বাইট কোড পড়ে সেগুলো মেশিন এক্সিকিউটেবল কোড-এ রূপান্তরিত করতে পারে। JVM অনেকগুলো মেশিনের জন্যে লেখা হয়েছে- অর্থাৎ এটি উইন্ডোজ, ম্যাক OS, লিনাক্স, আইবিএম mainframes, সোলারিস ইত্যাদি অপারেটিং সিস্টেমের জন্যে আলাদা আলাদা করে লেখা হয়েছে। এর ফলে, আমরা যদি একবার কোন জাভা প্রোগ্রাম লিখি, সেটি যেকোন মেশিনে চালাবো যাবে। এর কারণ আমরা এখন কোন নির্দিষ্ট মেশিনকে উদ্দেশ্য না করে শুধু মাত্র JVM কে উদ্দেশ্য করে কোড লিখি। যেহেতু সব মেশিনের জন্যেই JVM আছে, সুতরাং আমাদের কোড সব মেশিনেই চলবে। আর এভাবেই - **“Write once, run anywhere”** বা **WORA** সম্ভব হয়েছে।

৩. জাস্ট ইন টাইম( **JIT**) কম্পাইলার – এটি মূলত JVM এর একটি অংশ। আমরা যে জাভা কোড কম্পাইল করার সময় তৈরি করি সেগুলো মূলত JIT কম্পাইলার প্রসেস করে। একে dynamic translator ও বলা যায়- কারণ এটি রানটাইম-এ অর্থাৎ প্রোগ্রাম চলাকালীন সময়ে বাইটকোড প্রসেস করে।

এবার আমরা আরও কিছু টার্মিনোলজি(পরিভাষা) এর সাথে পরিচিত হই।

জাভা রানটাইম এনভায়রনমেন্ট (**JRE**) –এটি মূলত একটি জাভা প্রোগ্রাম রান করার জন্যে অত্যন্ত:পক্ষে যে সব কম্পোনেন্ট লাগে তার একটি প্যাকেজ। এর মধ্যে থাকে JVM এবং কিছু স্ট্যান্ডার্ড এপিআই।

জাভা ডেভেলপার কিট (**JDK**) – এটি হচ্ছে JRE এবং জাভা কোড লেখার জন্যে যে সব টুল গুলো লাগে তার একটি সেট। জাভা প্রোগ্রাম লেখার জন্য শুধু মাত্র JDK থাকলেই চলে কারণ এর মাঝেই সব কিছু দেয়া থাকে।

জাভার তিনটি সারসেট আছে সেগুলো হলো -

জাভা স্ট্যান্ডার্ড এডিশন (**JSE**)

- ডেব্রাটপ এবং স্ট্যান্ড-অ্যালোন সার্ভার এপ্লিকেশন তৈরি করার জন্যে যে সব টুল এবং এপিআই দরকার হয় সেগুলোকে আলাদা করে এর নাম দেওয়া হয়েছে জাভা স্ট্যান্ডার্ড এডিশন।

জাভা এন্টারপ্রাইস এডিশন (**JEE**) – এটি JSE এর উপর তৈরি ওয়েব এবং অনেক বড় মাপের এন্টারপ্রাইজ এপ্লিকেশন তৈরি করার জন্যে যে সব কম্পোনেন্ট দরকার হয় সেগুলোকে আলাদা করে এর নাম দেওয়া হয়েছে জাভা এন্টারপ্রাইস এডিশন- উদাহরণস্বরূপ এর কম্পোনেন্ট গুলো হচ্ছে-

- Servlets
- Java Server Pages (JSP)
- Java Server Faces (JSF)
- Enterprise Java Beans (EJB)
- Two-phase commit transactions
- Java Message Service message queue API's (JMS)
- etc.

জাভা মাইক্রো এডিশন (**JME**)

- এটি মূলত জাভা স্ট্যান্ডার্ড এডিশন এর সংক্ষিপ্ত এডিশন। ইন্টারনেট অব থিংস, এমবেড ডিভাইস, মোবাইল ডিভাইস, মাইক্রোকন্ট্রোলার, সেন্সর, গেটওয়ে, মোবাইল ফোন, ব্যক্তিগত ডিজিটাল সহায়ক (পিডিএ), টিভি সেট টপ বক্স, প্রিন্টার ইত্যাদি জন্যে তৈরি জাভার এই সংক্ষিপ্ত এডিশন কে বলা হয় - জাভা মাইক্রো এডিশন।

এবার তাহলে জাভা চলুন জাভা ইন্সটল করে ফেলি--

লিনাক্স মেশিনে জাভা ইন্সটল করতে নিচের ধাপ গুলো apply করতে হবে-

- ধাপ ১: নিচের লিংক থেকে জাভা ডাউনলোড করে নিন।

### Oracle JDK 7 Download Link

- ধাপ ২: এরপর টার্মিনাল থেকে যেখানে জাভা ডাউনলোড হয়েছে সেখানে যান-

```
cd ~/Download
```

- ধাপ ৩: এবার JDK ইন্সটল করি-

```
sudo tar -xzf jdk-7u21-linux-i586.tar.gz --directory=/usr/local/
```

```
sudo ln -s /usr/local/[jdk_folder_name]/ /usr/local/jdk
```

jdk\_folder\_name - আপনার পছন্দমত একটি নাম দিন।

- ধাপ ৪: আবার টার্মিনালে ফিরে যান- .bashrc অপেন করুন।

```
sudo gedit .bashrc
```

- ধাপ ৫ : .bashrc ফাইল-এ নিচের লাইনটি এড করুন।

```
export JAVA_HOME=/usr/local/jdk
```

Save and close .bashrc file.

- ধাপ ৬: কম্পাইল .bashrc ফাইল

```
source .bashrc
```

- ধাপ ৭: এবার পরীক্ষা করে দেখা যাক জাভা ইন্সটল হয়েছে কিনা। আবার টার্মিনাল ওপেন করুন এবং নিচের লাইনটি টাইপ করুন।

```
java -version
```

যদি সবকিছু ঠিকঠাক থাকে তাহলে আপনি নিচের তথ্য গুলো দেখতে পারবেন-

```
java version "1.7.0_65"
Java(TM) SE Runtime Environment (build 1.7.0_65-b17)
Java HotSpot(TM) 64-Bit Server VM (build 24.65-b04, mixed mode)
```

আর উইন্ডোজ মেশিনের ক্ষেত্রে এটি আরো সহজ। এর জন্যে শুধুমাত্র JDK টি ডাউনলোড করে ডাবল-ক্লিক করেই এটি ইন্সটল করা যাবে।

### IDE-

এক্ষেত্রে আমি দুটি আইডিইর কথা বলতে পারি-

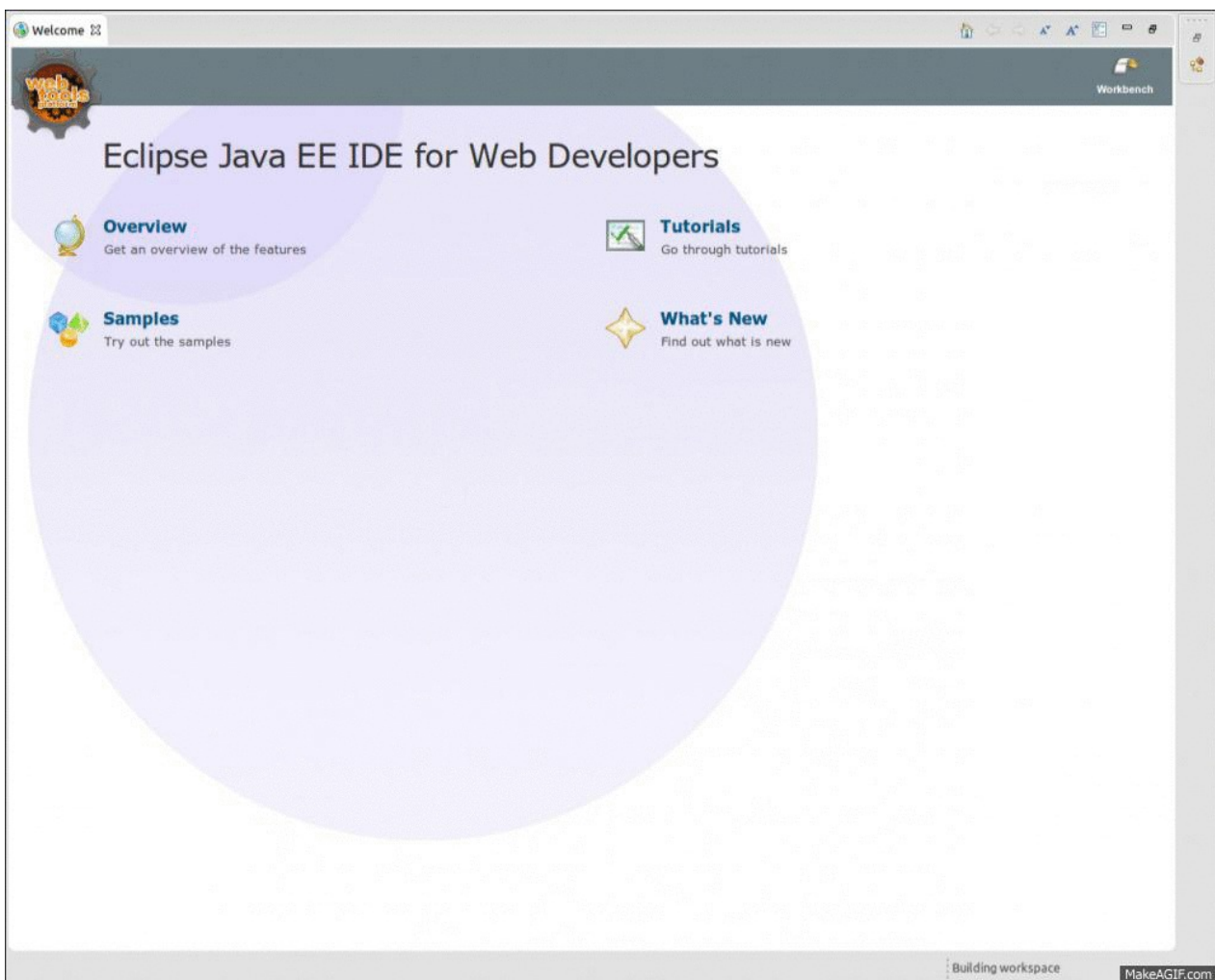
- ১. Eclipse - <https://www.eclipse.org/downloads/>

- ২. IntelliJ IDEA - <http://www.jetbrains.com/idea/download/>

তবে এই টিউটোরিয়ালে আমরা Eclipse ব্যবহার করবো।

তো চলুন- এবার তাহলে আমাদের প্রথম Hello world প্রোগ্রামটি লিখে ফেলি।

```
package bd.com.howtocode.java.helloworld;  
  
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```



## পাঠ ২: সিনট্যাক্স

- প্যাকেজ ডিক্লেয়ারেশন
- ইম্পোর্ট
- ক্লাস
- ফিল্ডস
- মেথডস
- কন্সট্রাকটরস
- কন্সটেন্টস

এই চ্যাপ্টারে আমি একটি জাভা প্রোগ্রাম এর মৌলিক কিছু ওভারভিউ দেয়ার চেষ্টা করবো। তবে শুরুতে সুবিধার্থে আমাদের কিছু টার্মস সম্পর্কে জেনে নেওয়া জরুরি।

### অবজেক্ট

যেহেতু জাভা একটি অবজেক্ট ওরিয়েন্টেড ল্যাংগুয়েজ, সুতরাং শুরুতে জানতে হবে অবজেক্ট কি। অবজেক্ট এর মানে আমরা যা জানি, সেটা হচ্ছে আমাদের জড়জগতের কোন বস্তু, যাকে ঠিক স্পর্শ করা যায়। তবে যেহেতু আমরা কল্পনা করতে পারি, আমরা অনেক কিছু ধরে নিতে পারি, মনে করুন - একটি বাইসাইকেল। বাইসাইকেল বলতেই আমাদের মাথায় একটি চিত্র চলে আসে। আমরা এর বৈশিষ্ট্যগুলো জানি, যেমন এটির দুইটি চাকা থাকে, একটি বসার সিট থাকে, এর ব্রেক আছে। তারপর এও জানি যে এটি কি করে, অর্থাৎ সাইকেল এর কাজ গুলোও আমরা জানি- যেমন এটি চলে। দেখা যাচ্ছে যে আমরা একটি বাইসাইকেল এর অবস্থা ও আচরণ সম্পর্কে জানি। এই অবস্থা ও আচরণ গুলো নিয়েই বাইসাইকেল একটি অবজেক্ট।

আমরা যদি আমাদের কল্পনাটুকু আরেকটু বাড়িয়ে নিয়ে বলি, সাইকেল হচ্ছে একটি সফটওয়্যার কম্পোনেন্ট যা কিনা কম্পিউটারে চলে, আমার মনে হয় কারো আপত্তি থাকার কথা নয়।

যেহেতু আমরা প্রোগ্রামিং নিয়ে আলোচনা করছি, সুতরাং এভাবে বলি, আমরা যদি একটা প্রোগ্রাম লিখি, সেই প্রোগ্রামের ছোট্ট একটি অংশ যার আমাদের এই বাইসাইকেল এর মতো বৈশিষ্ট্য থাকে, এবং একটি কিছু কাজ সম্পাদন করতে পারে, তাহলে সেই ছোট্ট অংশটিকে অবজেক্ট বলতে পারি।

### ক্লাস

মনে করি আমরা একটা বাড়ি বানাতে চাই। প্রথমে আমরা চিন্তা করি বাড়িটা আসলে কিভাবে বানাবো। আমরা জায়গা নির্বাচন করি। তারপর চিন্তা করি বাড়িটি কত-তলা হবে, কয়টা এপার্টমেন্ট হবে, এপার্টমেন্ট গুলো কত স্কয়ারফিটের হবে। তারপর চিন্তা করি, একটা এপার্টমেন্ট এ কয়টি রুম হবে, ড্রয়িং রুমের দৈর্ঘ্য কত হবে, কয়টা বাথ থাকবে, বেলকনি কোথায় থাকবে, রান্না ঘর কোথায় হবে ইত্যাদি ইত্যাদি। আচ্ছা এগুলো ঠিক হয়ে গেল, এখন আমরা চিন্তা করবো আরও জটিল কাজ নিয়ে। ওয়্যারিং নিয়ে, প্রত্যেক রুমে কয়টা পয়েন্ট থাকবে, পানির লাইন কিভাবে নেব। তারপরে বাথরুমে কি ধরনের টাইল ব্যবহার করবো, ফ্লোরে কোন গুলো।

অর্থাৎ বাড়িটি বানানোর আগেই আমরা সব কিছু নির্ধারণ করে ফেলছি এবং আমরা এই বিষয়গুলো সব লিপিবদ্ধ করে রাখি। তারপর এই লিপিবদ্ধ লেখাগুলোকে নানাভাবে পরীক্ষা করে ক্রস চেক করে চূড়ান্ত করি। এর একটি গলাভরা নাম আছে, সেটা হচ্ছে- blueprint.

আমাদের এক্ষেত্র বাড়তি হচ্ছে অবজেক্ট। এই অবজেক্ট বানানোর আগে আমাদের blueprint এর দরকার হয়। আর এই blueprint কেই আমরা বলি ক্লাস।

আমরা তাহলে এখন অবজেক্ট এবং ক্লাস এর ধারণা জানি। এবার তাহলে আমাদের মূল বিষয় সিনট্যাক্স নিয়ে কথা বলি-  
আমরা যারা সি কিংবা অন্য কোন প্রোগ্রামিং ল্যাংগুয়েজ আগে থেকেই জানি, একটি প্রোগ্রামে দুটি জিনিস অবশ্যই কমত থাকে - সেগুলো হলো - ফাংশান এবং ডেটা।

একটি জাভা প্রোগ্রাম লিখতে হলে আমাদেরকে অবশ্যই একটি ফাইল তৈরি করতে হবে যার এক্সটেনশন হবে .java.  
উদাহরণস্বরূপ- HelloWorld.java এবার আমরা লক্ষ্য করি একটি জাভা প্রোগ্রামে কি কি থাকে-

- প্যাকেজ ডিক্লারেশন
- ইম্পোর্ট স্টেটমেন্টস
- টাইপ ডিক্লারেশন
  - ফিল্ডস
  - মেথডস

উপরের নামগুলো নিয়ে দৃষ্ট লাগলে সমস্যা নেই, এক্ষণি সেগুলো নিয়ে আলোচনা করছি, তবে তার আগে একটি জাভা প্রোগ্রাম দেখে নিই।

```
package bd.com.howtocode.java.tutorial.syntax;

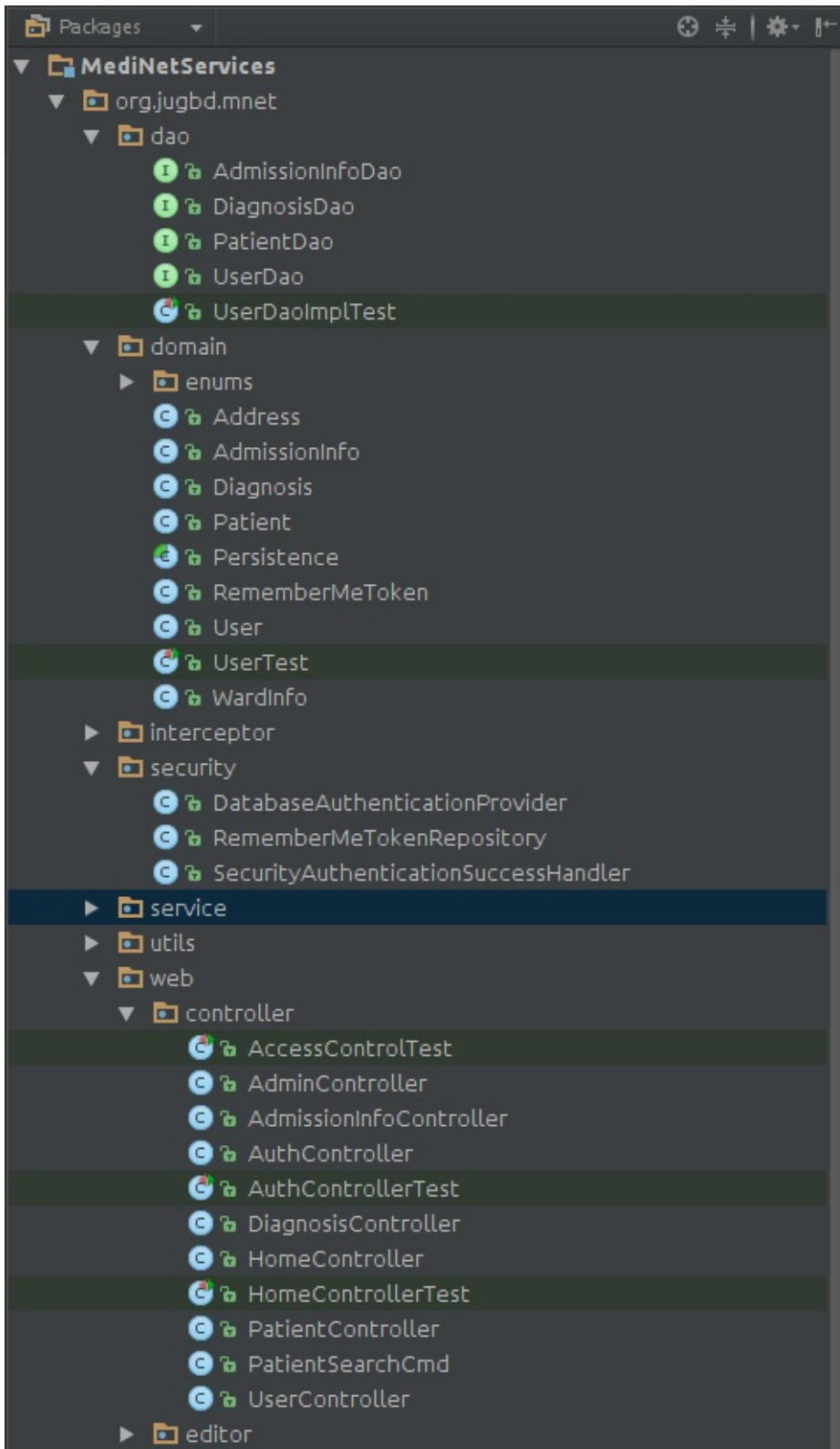
import java.util.HashMap;

public class HelloWorld {
    protected final String hello = "value";

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

এই কোডটির শুরুতেই আছে প্যাকেজ ডিক্লারেশন। আমরা আমাদের কম্পিউটারে নানা ধরনের ফাইল বিভিন্ন ফোল্ডারে সাজিয়ে রাখি। যেমন- মুভি ফোল্ডারে হয়তো আমরা শুধুই মুভি রাখি, সেখানে অন্য ফাইল রাখি না। আবার মুভি ফোল্ডারে এর মধ্যে আরো সাব-ফোল্ডার তৈরি করি আরো আলাদা করার জন্যে, যেমন – বাংলা মুভি, ইংরেজি মুভি ইত্যাদি। জাভাতে প্যাকেজ বলতে এই ধারণাটাই বুঝায়। একটি জাভা প্রোগ্রামিং ভাষায় লেখা সফটওয়্যার এ শত শত বা হাজার হাজার পৃথক ক্লাস থাকতে পারে। এজন্যে প্যাকেজ ডিক্লারেশন এর মাধ্যমে আমরা একি রকম ক্লাস গুলো একটি প্যাকেজের মধ্যে আলাদা করে রাখি।

উদাহরণস্বরূপ এখানে প্যাকেজ স্ট্রাকচার এর একটি স্ক্রিনশট দেওয়া হল-



প্যাকেজ নাম গুলোকে লোয়ার কেস অস্করে-এ লিখতে হয়।

কোম্পানি গুলো তাদের ইন্টারনেট ডোমেইন নেইম কে উল্টো করে তাদের প্যাকেজের নাম লিখে। যেমন - example.com এর একটি প্রোগ্রামার একটি প্যাকেজের নাম লিখবে এইভাবে- com.example.package.

আমাদের ক্ষেত্রে-

```
package bd.com.howtocode.java.tutorial.syntax;
```

তারপর আমাদের প্রোগ্রামের দ্বিতীয় লাইনটি হলো - ইম্পোর্ট স্টেটমেন্টস। অন্য কোন প্যাকেজের ক্লাস যদি আমাদের প্রোগ্রামে দরকার হয় তাহলে আমরা সেটিকে এভাবে ইম্পোর্ট করতে পারি। এটি সি প্রোগ্রামিং এর ইনক্লুড স্টেটমেন্টস এর মতো।

```
import java.util.HashMap;
```

এর পরের লাইনটি হলো টাইপ ডিক্লারেশন। জাভাতে একটি টাইপ একটা ক্লাস অথবা ইন্টারফেস অথবা এনাম হতে পারে (ইন্টারফেস এবং এনাম নিয়ে পরে আলোচনা করা হবে)। ক্লাস ক্ষেত্রে শুরুতে class কিওয়ার্ড লিখেতে হয় তারপর কার্লি ব্রেস { শুরু এবং শেষ } করতে হয়। আমাদের পরবর্তি প্রতিটা লাইন কোড এই কার্লি ব্রেস { } এর ভেতরে লিখতে হবে।

```
public class HelloWorld { }
```

এখানে অতিরিক্ত একটি public কিওয়ার্ড দেখা যাচ্ছে। এই মুহুর্তে শুধু মনে রাখুন ক্লাস এর শুরুতে এটি লিখতে হয়। পরে এটি নিয়ে আলোচনা করা হবে।

এর পরেই আমরা যা দেখছি তাকে বলা হয় ফিল্ড ডিক্লারেশন। অর্থাৎ আমরা যে বিভিন্ন রকম ভ্যারিয়েবল ডিক্লার করি, সেগুলো।

```
protected final String hello = "value";
```

এবং এর পরেই থাকে মেথড। সি কিংবা অন্যান্য প্রোগ্রামিং ল্যাংগুয়েজ এ যাকে আমরা ফাংশন কিংবা সাবরুটিন বলে থাকে, এখানে আমরা সেগুলোকে মেথড বলি।

এক্ষেত্রে আমাদের মেথড হচ্ছে -

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

এটি হচ্ছে মেইন মেথড। জাভা প্রোগ্রামকে রান করতে হলে অবশ্যই কোন ক্লাসে একটি মেইন মেথড থাকতে হবে। এবার আমরা কিছু জিনিস প্রিন্ট করার চেষ্টা করি-

জাভাতে কনসলে কিছু প্রিন্ট করার জন্যে System.out.println() অথবা System.out.print() ব্যবহার করা হয়।

আমরা যদি নিচের প্রোগ্রামটি রান করি-



```
package bd.com.howtocode.java.tutorial.syntax;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, world!"); // Advance the cursor to the beginning
        System.out.println();                // Print a empty line
        System.out.print("Hello, world!");    // Cursor stayed after the printed s
        System.out.println("Hello,");
        System.out.print(" ");                // Print a space
        System.out.print("world!");
        System.out.println("Hello, world!");
    }
}
```

তাহলে কনসলে নিচের লাইন গুলো প্রিন্ট হবে-

```
Hello, world!

Hello, world!Hello,
world!Hello, world!
```

আমরা ইতিমধ্যে জানি ক্লাস কি- তাহলে এবার একটি ক্লাস লিখে ফেলা যাক-

```
package bd.com.howtocode.java.tutotorial.syntax;

/**
 * @author Bazlur Rahman Rokon
 * @since 9/20/14.
 */
public class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:" +
            cadence + " speed:" +
            speed + " gear:" + gear);
    }
}
```

আমরা ক্লাস এবং অবজেক্ট কি জানি, কিন্তু কিভাবে ক্লাস থেকে অবজেক্ট তৈরি করতে হয় সেটি এবার দেখা যাক-

```

package bd.com.howtocode.java.tutorial.syntax;

/**
 * @author Bazlur Rahman Rokon
 * @since 9/20/14.
 */

public class BicycleDemo {
    public static void main(String[] args) {
        // Create two different
        // Bicycle objects

        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}

```

আমরা জানি যে জাভা প্রোগ্রাম চালু করতে হলে একটি মেইন মেথড দরকার হয়। উপরের প্রোগ্রামটিতে একটি মেইন মেথড আছে। এবং এর ভেতরে শুরুতে আমরা দুইটি অবজেক্ট তৈরি করেছি।

```

Bicycle bike1 = new Bicycle();
Bicycle bike2 = new Bicycle();

```

জাভাতে অবজেক্ট তৈরি করা খুব সহজ। এর জন্যে আমাদের তিনটি স্টেপ দরকার হয়-

- ডিক্লারেশন
- ইনস্ট্যানশিয়েশন
- ইনিশিয়ালাইজেশন

**Bicycle bike1 = new Bicycle();**

উপরের বোল্ড অক্ষরে লেখাটুকু হচ্ছে ডিক্লারেশন, তারপর সমান চিহ্ন এর পর new কিওয়ার্ড পর্যন্ত হচ্ছে ইনস্ট্যান্সিয়েশন এবং এর পরের অংশটুকুকে ইনিশিয়ালাইজেশন বলা হয়। ইনিশিয়ালাইজেশন এর জন্য আমাদের ক্লাসটির কনস্ট্রাকটরকে কল করতে হয়। কনস্ট্রাকটর নিয়ে একটু পরেই কথা বলছি।

এখানে ডিক্লারেশন টাইপ ডিক্লারেশন এর মতোই। ড্যারিয়বল চ্যাপ্টারে আমরা আরো ডিটেইলস দেখবো।

তারপর অবজেক্টটি ধরে ডট অপারেটর ব্যবহার করে সেই ক্লাসের মেথড গুলো কল করা হয়েছে। এই প্রোগ্রামটি রান করলে আউটপুট আসবে-

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

### কনস্ট্রাকটর

কনস্ট্রাকটর অন্যান্য মেথড বা ফাংশনের মতই একটি মেথড বা ফাংশনে। তবে এটির কোন রিটার্ন টাইপ নেই। একটি ক্লাসকে একটি অবজেক্ট-এ তৈরি করতে যে প্রয়োজনীয় কাজ গুলো করতে হয়, কনস্ট্রাকটর সেই কাজ গুলো করে থাকে। তবে মজার ব্যাপার হচ্ছে সেই প্রয়োজনীয় কাজ গুলো জন্য আমাদের কোড লিখতে হয় না।

আমাদের উপরের ক্লাসটিতে আমরা কোন কনস্ট্রাকটর লিখি নি। তাহলে এর অবজেক্ট তৈরি হলো কিভাবে? উত্তরটি হচ্ছে আমরা যদি কোন কনস্ট্রাকটর না লিখি তাহলে জাভা কম্পাইলার নিজে থেকেই একটি কনস্ট্রাকটর লিখে কম্পাইল করে, যাকে আমরা বলি ডিফল্ট কনস্ট্রাকটর। তবে আমরা চাইলে নিজের একটি লিখতে পারি।

```
public class Bicycle {
    int cadence = 0;
    int speed = 0;
    int gear = 1;

    public Bicycle() {
    }
}
```

এবার আমরা দেখবো কিভাবে জাভাতে কমেন্ট লিখতে হয়-

জাভা তিন ধরনের কমেন্ট সাপোর্ট করে-

Comment	Description
<code>/* text */</code>	জাভা কম্পাইনার
<code>// text</code>	জাভা কম্পাইনার
<code>/** documentation */</code>	এটি হচ্ছে ডকুমেন্টেশন কমেন্ট। একে d

উদাহরণ-

```

package bd.com.howtocode.java.tutotorial.syntax;

/**
 * The HelloWorld program implements an application that
 * simply displays "Hello World!" to the standard output.
 *
 * @author Bazlur Rahman Rokon
 * @since 9/20/14.
 */
public class HelloWorld {
    public static void main(String[] args) {
        // Prints Hello, World! on standard output.
        System.out.println("Hello, world!");

        /*
        for (int i = 0; i < 100; i++) {
            System.out.println(i);
        }*/
    }
}

```

আরও কিছু নিয়ম:

- জাভাতে প্রত্যেকটি স্টেটমেন্ট এর পর সেমিকোলন (;) দিয়ে স্টেটমেন্ট শেষ করতে হয়।
- জাভা একটি কেইস সেনসিটিভ ল্যাংগুয়েজ- অর্থাৎ hello এবং Hello দুটি আলাদা শব্দ।

অনুশীলন:

নিচের প্যাটার্নগুলো প্রিন্ট করতে চেষ্টা করুন -

* * * * *	* * * * *	* * * * *
* * * * *	* * * * *	* * * * *
* * * * *	* * * * *	* * * * *
* * * * *	* * * * *	* * * * *
* * * * *	* * * * *	* * * * *
(a)	(b)	(c)

## পাঠ ৩: ডাটা টাইপস এবং অপারেটর

- ভেরিয়েবল
- প্রিমিটিভ ডাটাটাইপ, ইন্টিজার, লং, ডাবল, ইন্টিজার, ফ্লোট এবং কার
- রপার ক্লাস
- লিটারেল
- বিভিন্ন রকম অপারেটর

### ভ্যারিয়েবল

ভ্যারিয়েবল হচ্ছে একটি নাম যা কম্পিউটারের একটি মেমোরি লোকেশন কে নির্দেশ করে। উদাহরণ-

```
int cadence = 0;
```

একটি ভ্যারিয়েবল ডিক্লারেশন এর জন্যে একটি ডাটাটাইপ দরকার হয়, অর্থাৎ ভ্যারিয়েবল টি কি ধরনের ডাটা হোল্ড করতে তা বলে দিতে হবে। উপরের উদাহরণটিতে আমরা একটি ভ্যারিয়েবল ডিক্লার করেছি যার নাম cadence এবং এটি ইন্টিজার টাইপ ডাটা হোল্ড করে।

যেহেতু জাভা একটি স্ট্যাটিক্যালি টাইপড ল্যাংগুয়েজ সুতরাং ভ্যারিয়েবল ডিক্লারেশন এর সময় ডাটাটাইপ উল্লেখ করা অত্যাবশ্যক।

জাভাতে আমরা চার ধরনের ভেরিয়েবল নিয়ে কাজ করে থাকি -

1. Instance Variables (Non-static fields)
2. Class Variables (Static Fields)
3. Local variables
4. Parameters variables

জাভাতে ভ্যারিয়েবল এবং ফিল্ড দুই শব্দই ব্যবহার করা হয়, তবে এর কিছু টেকনিকাল পার্থক্য আছে। সেগুলো নিয়েই আলোচনা করা হবে -

আমরা আবার একটি উদাহরণ দেখি -

```

public class Bicycle {
    static int numGears = 6;

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    public Bicycle() {
    }

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:" +
            cadence + " speed:" +
            speed + " gear:" + gear);
    }
}

```

আমরা জানি যে একটি ক্লাস থেকে আমরা অবজেক্ট তৈরি করি। আমরা একটা ক্লাস থেকে অনেকগুলো অবজেক্ট তৈরি করতে পারি। এবং প্রত্যেক অবজেক্ট-ই আলাদা আলাদা। যেমন -

```

Bicycle bike1 = new Bicycle();
Bicycle bike2 = new Bicycle();

```

এখানে bike1 এবং bike2 দুটি সম্পূর্ণ আলাদা অবজেক্ট।

এখন bike1 এবং bike2 তে কিছু ভ্যারিয়েবল গুলোও আলাদা। অর্থাৎ আমরা যতগুলো অবজেক্ট তৈরি করতেরা ঠিক ততোগুলো আলাদা ভ্যারিয়েবল থাকবে মেমোরিতে। এক্ষেত্রে মেমোরিতে ২টা cadence থাকবে, ২টা gear থাকবে এবং ২ speed থাকবে।

এই ভ্যারিয়েবল গুলোকে Instance Variables বা Non-static fields বলা হয়। এই ভ্যারিয়েবল গুলো আগে static কিওয়ার্ডটি থাকে না।

```
static int numGears = 6;
```

উপরের উদাহরণটিতে **numGears** নামে একটি ভ্যারিয়েবল আছে, এটির আগে একটি **static** কিওয়ার্ডটি আছে। এ ধরনের ভ্যারিয়েবল কে Class Variables বা Static Fields বলা হয়। static কিওয়ার্ডটি কম্পাইলারকে বলে যে numGears নামে একটি মাত্র ভ্যারিয়েবল থাকবে মেমোরিতে, অবজেক্ট এর সংখ্যা যতই হোক।

লোকাল ভ্যারিয়েবল হলো সেসব ভ্যারিয়েবল যে গুলো কোন মেথডের মাঝে ডিক্লার করা হয়। একটি লোকাল ভ্যারিয়েবল শুধু মাত্র সেই মেথডের ভেতর থেকেই একসেস করা যাবে।

আর Parameters variables হলো সেই ভ্যারিয়েবল গুলো যেগুলো মেথড কল করার সময় পাস করা হয়। এ গুলোও শুধুমাত্র মেথডের ভেতর থেকেই একসেস করা যায়।

আমরা Instance Variables এবং Class Variables গুলোকে ফিল্ড বলি।

এখানে কিছু ভ্যারিয়েবল ডিক্লারেশনের উদাহরণ দেওয়া হলো -

```
byte    myByte;
short   myShort;
char    myChar;
int     myInt;
long    myLong;
float    myFloat;
double  myDouble;
```

শুরুতে আগে টাইপ লিখতে হবে, তারপর একটি নাম, তারপর সেমিকোলন দিয়ে শেষ করতে হবে। তবে আমরা চাইলে ভ্যারিয়েবল কে ইনিশিয়ালাইজেশান করতে পারি। যেমন -

```
int cadence = 0;
```

অর্থাৎ শুরুতে আমরা cadence এর ভ্যালু 0 এসাইন করলাম।

এরপর যদি আমরা কোন ভ্যারিয়েবলে ভ্যালু এসাইন করতে চাই তাহলে -

```
myByte    = 127;
myFloat    = 199.99;
```

জাভা ভ্যারিয়েবল লেখার কিছু নিয়ম কানুন আছে-

1. ভ্যারিয়েবল গুলো কেইস সেনসিটিভ। অর্থাৎ money, Money, MONEY তিনটি আলাদা।
2. ভ্যারিয়েবল অবশ্যই যেকোন একটি লেটার দিয়ে শুরু করতে হবে। তবে \$ অথবা \_ দিয়েও শুরু করা যায়।
3. ভ্যারিয়েবল এর মাঝে নাম্বার কিংবা \_ থাকতে পারে।
4. ভ্যারিয়েবল জাভার কোন reserved কিওয়ার্ড হতে পারবে না।

ডাটা টাইপ



জাভা তে আট ধরনের প্রিমিটিভ ডাটা টাইপ আছে ।

Data type	Description
byte	8 bit signed value, values from -128 to 127
short	16 bit signed value, values from -32.768 to 32.767
char	16 bit Unicode character
int	32 bit signed value, values from -2.147.483.648 to 2.147.483.647
long	64 bit signed value, values from -9.223.372.036.854.775.808 to 9.223.372.036.854.775.808
float	32 bit floating point value
double	64 bit floating point value

এগুলো প্রিমিটিভ , এর মানে হচ্ছে এগুলো অবজেক্ট নয় । এরা মেমোরিতে সরাসরি ড্যালু রাখে ।

রেপার ক্লাস

তবে জাভাতে কিছু ডাটা টাইপ আছে যেগুলো অবজেক্ট ।

Data type	Description
Byte	8 bit signed value, values from -128 to 127
Short	16 bit signed value, values from -32.768 to 32.767
Character	16 bit Unicode character
Integer	32 bit signed value, values from -2.147.483.648 to 2.147.483.647
Long	64 bit signed value, values from -9.223.372.036.854.775.808 to 9.223.372.036.854.775.808
Float	32 bit floating point value
Double	64 bit floating point value

এগুলোকে প্রিমিটিভ টাইপ এর রেপার রেপার ক্লাস বলা হয় । লক্ষ্য করণ, এগুলোর সবগুলোর নাম ক্যাপিটাল অক্ষর দিয়ে শুরু হয়েছে ।

তবে আমরা চাইলে অবজেক্ট ডাটাটাইপ এবং প্রিমিটিভ ডাটাটাইপ একে অপরের পরিপূরক হিসাবে ব্যবহার করতে পারি ।

```
Integer a;
int b = 9;
a = b;
```

তবে প্রিমিটিভ ভ্যালু গুলো ডিফল্ট ভ্যালু থাকে। অর্থাৎ আমরা যদি ভ্যালু এসাইন না করি, তাহলে এদের মধ্যে বাইডিফল্ট ভ্যালু থাকে। যেমন -

Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

লিটারেল-

প্রোগ্রামিং ল্যাংগুয়েজ কিছু মজার মজার বিল্ট-ইন সুবিধা থাকে, তার মধ্যে লিটারেল একটি। আমরা জানি যে একটা ভ্যারিয়েবল ডিক্লারেশন এর জন্য প্রথমে টাইপ লিখতে হয়, তারপর একটা নাম দিতে হয়, তারপর একে ইনিশিয়ালাইজেশন করতে হয়। ভেরিয়েবলটি যদি অবজেক্ট হয়, তাহলে ইনটেনশিয়েশন করতে হয়।

উদাহরণ-

```
List list = new ArrayList();

or

Int x = 5;
```

উপরের দুটি উদাহরণের মাঝে একটিতে আমরা new কিওয়ার্ড ব্যবহার করে নতুন অবজেক্ট তৈরি করেছি। কিন্তু পরের উদাহরণটিতে সেটি করতে হয় নি। আমরা সরাসরি একটি ভ্যালু এসাইন করেছি। এখানে 5 একটি ভ্যালু। এখানে 5 হচ্ছে লিটারেল।

এরকম অনেক ক্ষেত্রে আমরা new কিওয়ার্ড ব্যবহার না করেই ভেরিয়েবল initialize করতে পারি।

জাভাতে প্রিমিটিভ টাইপ সকল ডাটাটাইপ লিটারেল সাপোর্ট করে। যেমন -

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

নিচে আরো কিছু উদাহরণ দেওয়া হলো –

## ইন্টিজার লিটারেল-

```
// এখানে 26 হচ্ছে ডেসিমাল নাম্বার
int decVal = 26;
// এখানে 26 গ্রন্থ্যটি হেক্সাডেসিমেল এ দেখানো হয়েছে
int hexVal = 0x1a;
// এখানে 26 গ্রন্থ্যটি বাইনারি-তে এ দেখানো হয়েছে
int binVal = 0b11010;
```

## ফ্লোটিং পয়েন্ট লিটারেল-

```
double d1 = 123.4;
// একি ভ্যানু বৈজ্ঞানিক উপায়ে লেখা হয়েছে
double d2 = 1.234e2;
float f1 = 123.4f;
ক্যারেঞ্জার এন্ড স্ট্রিং লিটারেল--
```

char এবং String উদ্ধৃতি চিহ্নের ভেতরে লেখা হয়। char ক্ষেত্রে একক উদ্ধৃতি চিহ্ন String এর জন্যে ডবল উদ্ধৃতি চিহ্ন ব্যবহার করতে হয়- যেমন-

```
char chr = 'A'; // ক্যারেঞ্জার লিটারেল
String name = "Bazlur"; // স্ট্রিং লিটারেল
```

char এবং String ইউনিকোড ক্যারেঞ্জার হতে পারে।

আমরা জানি কিভাবে ভেরিয়েবল ইনিশিয়ালাইজ করতে হয় জানি, এবার তাহলে এই ভ্যারিয়েবল গুলো দিয়ে কি কাজ করা যায় সেগুলো দেখি।

কোন কাজ করতে হলে একজন কার্যকারী বা অপারেটর লাগে। অপারেটর কিছু অপারেশন নিয়ে কাজ করে থাকে তারপর ফলাফল রিটার্ন করে। জাভা প্রোগ্রামিং ল্যাংগুয়েজ এ বেশ কিছু অপারেটর আছে- সেগুলো দেখা যাক-

## এসাইনমেন্ট অপারেটর (Assignment Operator)

“=” এটি হচ্ছে এসাইনমেন্ট অপারেটর বাংলায় যাকে বলে সমান সমান চিহ্ন। আমরা একটি Bicycle ক্লাস দেখেছি, এর মাঝে কিছু ভেরিয়েবল দেখেছি-

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

এই ভ্যারিয়েবল গুলোর ডান পাশে সমান সমান চিহ্নের পর আমরা একটা ভ্যালু বা মান বসিয়েছি। এভাবে আমরা একটি ভ্যারিয়েবল এর মাঝে ভ্যালু এসাইন করতে পারি।

## এরিথমেটিক অপারেটর(Arithmetic Operator)

জাভা প্রোগ্রামিং ল্যাংগুয়েজ-এ যোগ, বিয়োগ, গুন, ভাগ করার জন্যে কিছু অপারেটর আছে। এগুলো আমরা যখন বেসিক গণিত শিখি তখন থেকেই জানি। শুধু একটি অপারেটর নতুন মনে হতে পারে, যা হলো “%”। এটিকে অনেকেই পারসেন্টেজ বা শতকরা চিহ্ন হিসেবে ভুল করতে পারে, কিন্তু এটি আসলে তা নয়। এটি মূলত একটি সংখ্যাকে আরেকটি সংখ্যা দ্বারা ভাগ করে ভাগশেষ রিটার্ন করে।

অপারেটর	এর কাজ
+	আডিটিভ(Additive) অপারেটর, যা দুটি সংখ্যা বা স্ট্রিং যোগ করার জন্যে ব্যবহার করা হয়।
-	সাবট্রাকশান (Subtraction) অপারেটর যা একটি সংখ্যা থেকে আরেকটি সংখ্যা বিয়োগ করার জন্যে ব্যবহার করা হয়।
*	মাল্টিপ্লিকেশান (Multiplication) অপারেটর যা দুটি সংখ্যাকে গুন করে।
/	ডিভিশান(Division) অপারেটর, যা দিয়ে একটি সংখ্যাকে আরেকটি সংখ্যাকে ভাগ করা যায়।
%	রিমাইন্ডার (Remainder) অপারেটর যা একটি সংখ্যাকে আরেকটি সংখ্যা দ্বারা ভাগ করে ভাগশেষ রিটার্ন করে।

```

class ArithmeticDemo {

    public static void main (String[] args) {

        int result = 1 + 2;
        // এখানে result এর মান হচ্ছে 3
        System.out.println("1 + 2 = " + result);
        int original_result = result;

        result = result - 1;
        //এখানে result থেকে ১ ঘাটসুটাই করায় এর মান ২
        System.out.println(original_result + " - 1 = " + result);
        original_result = result;

        result = result * 2;
        // এখানে result এর সাথে ২ মাল্টিপ্লাই করার ফলে এর মান 4
        System.out.println(original_result + " * 2 = " + result);
        original_result = result;

        result = result / 2;
        //আবার result ডিভাইড করার ফলে এর মান হয়ে গেল 2
        System.out.println(original_result + " / 2 = " + result);
        original_result = result;

        result = result + 8;
        // ৮ যোগ করার ফলে এর result হলো 10
        System.out.println(original_result + " + 8 = " + result);
        original_result = result;

        result = result % 7;
        // result এর সাথে ৭ রিমাইন্ডার অপারেটর ব্যবহার করার ফলে এর মান হয়ে গেল 3, কারণ এটি শূন্য
        System.out.println(original_result + " % 7 = " + result);

    }
}

```

এই প্রোগ্রামটি রান করলে নিচের ফলাফল প্রকাশিত হবে।

```

1 + 2 = 3
3 - 1 = 2
2 * 2 = 4
4 / 2 = 2
2 + 8 = 10
10 % 7 = 3

```

### ইউনারি (Unary) অপারেটর

উপরের সব অপারেটর এর জন্যে আমাদের দুটি করে অপারেণ্ড দরকার হতো, তবে এই অপারেটর এর লাগে একটি।

এগুলো বিভিন্ন ধরনের কাজ করে থাকে যেমন – এক করে ইনক্রিমেন্টিং/ডিক্রিমেন্টিং বা একটা এক্সপ্রেশান নেগেট করা বা একটা বুলিয়ান-কে ইনভার্ট করা । এগুলো হল -, +, -, ++, --, ! উদাহরণ -

```
class UnaryDemo {

    public static void main(String[] args) {

        int result = +1;
        // এটি এক করে ইনক্রিমেন্ট করে, সুতরাং এখানে result এর মান 1
        System.out.println(result);

        result--;
        // এটি এক করে ডিক্রিমেন্ট করে, সুতরাং এখানে result এর মান 0
        System.out.println(result);

        result++;
        // এটি এক করে ইনক্রিমেন্ট করে, সুতরাং এখানে result এর মান আবার ১
        System.out.println(result);

        result = -result;
        // এখানে result কে নেগেট করে, সুতরাং এর মান এখন -1
        System.out.println(result);

        boolean success = false;
        // এখানে বুলিয়ানের মান হচ্ছে false
        System.out.println(success);
        // কিন্তু এর আগে একটি নেগেট অপারেটর এড করলে এটি হয়ে যায়
        System.out.println(!success);
    }
}
```

### ইকুয়ালিটি (Equality) এবং রেশনাল(Relational) অপারেটরস

ইকুয়ালিটি (Equality) এবং রেশনাল(Relational) অপারেটর গুলো নির্ধারণ করে একটি ভ্যালু অন্যটি থেকে বড় বা ছোট কিনা ।

```
==    দুটি ভ্যালু সমান হলে এই এক্সপ্রেশান এর মান true হয়
!=    দুটি ভ্যালু সমান না হলে true হয়
>     প্রথম ভ্যালু পরের ভ্যালু থেকে বড় হলে true হয়
>=    প্রথম ভ্যালু পরের ভ্যালু থেকে বড় বা সমান হলে true হয়
<     প্রথম ভ্যালু পরের ভ্যালু থেকে ছোট হলে true হয়
<=    প্রথম ভ্যালু পরের ভ্যালু থেকে ছোট বা সমান হলে true হয়
```

### উদাহরণ

```

class ComparisonDemo {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if(value1 == value2)
            System.out.println("value1 == value2");
        if(value1 != value2)
            System.out.println("value1 != value2");
        if(value1 > value2)
            System.out.println("value1 > value2");
        if(value1 < value2)
            System.out.println("value1 < value2");
        if(value1 <= value2)
            System.out.println("value1 <= value2");
    }
}

```

### কন্ডিশনাল( Conditional) অপারেটর

&& এবং || এই দুই অপারেটরকে কন্ডিশনাল অপারেটর বলে।

&&	কন্ডিশনাল অ্যান্ড (Conditional-AND)
	কন্ডিশনাল অর ( Conditional-OR)

### উদাহরণ-

```

class ConditionalDemo1 {

    public static void main(String[] args){
        int value1 = 1;
        int value2 = 2;
        if((value1 == 1) && (value2 == 2))
            System.out.println("value1 is 1 AND value2 is 2");
        if((value1 == 1) || (value2 == 1))
            System.out.println("value1 is 1 OR value2 is 1");
    }
}

```

### চলবে --

## ## এরে (Array)

## এরে

এরে হচ্ছে একধরনের কন্টেইনার অবজেক্ট যা অনেকগুলো একিধরনের ডাটা টাইপের এর একটি ফিক্সড সাইজের ড্যালু ধরে রাখতে পারে।

এরে ডিক্লার করার জন্যে প্রথমে ডাটাটাইপ (কি ধরনের ডাটাটাইপ রাখবে) এর সাথে ([]) স্বয়ার ব্র্যাকেট তারপর এর একটি ডেরিয়েবল নাম দিতে হয়।

জাভাতে দুই ধরনের এরে রয়েছে ১/ সিঙ্গেল ডাইমেনশনাল এরে ২/ মাল্টিডাইমেনশনাল এরে

সিঙ্গেল ডাইমেনশনাল এরে কে আমরা এইভাবে ডিক্লিয়ার করতে পারিঃ

```
//একটি ইন্টিজার এর  
int[] anArray;
```

তবে স্বয়ার ব্র্যাকেট ডেরিয়েবল নাম এর পরেও দেওয়া যেতে পারে - উদহরণ-

```
int anArray[];
```

এভাবে আমরা অন্য ডাটাটাইপ এর অ্যারে লিখতে পারি -

```
long[] anArrayOfLongs;  
float[] anArrayOfFloats;  
double[] anArrayOfDoubles;
```

এরে একটি অবজেক্ট, সুতরাং একে নিউ(new) অপারেটর দিয়ে প্রথমে ক্রিয়েট করতে হবে।

```
// এখানে ১০ মাইজের একটি এর ক্রিয়েট করা হলো  
anArray = new int[10];
```

এই স্ট্যাটমেন্ট যদি না লেখা হয় তাহলে প্রোগ্রামটি কম্পাইল হবে না।

এরপর আমরা এর এর ভেতর ড্যালু রাখতে পারি।

```
anArray[0] = 100; //এখানে প্রথম ভ্যালু রাখা হল  
anArray[1] = 200; // এভাবে দ্বিতীয় ভ্যালু  
anArray[2] = 300; // এভাবে বাকি গুনো
```

জাভা প্রোগ্রামিং ল্যান্গুয়েজে জিরো বেজড নাম্বারিং( ইনডেক্স শূন্য থেকে শুরু ) করা হয়ে থাকে। অর্থাৎ, এরের এই ড্যালুগুলো যদি পড়তে চাই তাহলে -



```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

এছাড়াও এরে লেখার শর্টকাট পদ্ধতি আছে -

```
int[] anArray = {
    100, 200, 300,
    400, 500, 600,
    700, 800, 900, 1000
};
```

একটি এরে এর মধ্যে আরেকটি অ্যারে রাখা যেতে পারে - নিচে মাল্টিডাইমেনশনাল এরে এর ডিক্লারেশন এবং ড্যালু এসাইন করার একটি পদ্ধতি দেখানো হল ।

```
int[][] anArray = {{1, 2, 3}, {4, 6, 7}, {8, 9}};
```

আমরা যদি একটি এরে এর লেন্থ বা সাইজ জানতে চাই তাহলে –

```
int length = anArray.length;
```

সুবিধাঃ এরেতে আমরা খুব সহজে ডেটা গুলোকে ছোট থেকে বড় - বড় থেকে ছোট আকারে সাজিয়ে নিতে পারি । খুব সহজেই যেকোনো ইনডেক্সে এক্সেস নিতে পারি । অসুবিধাঃ এরের সাইজ আগে থেকে বলে দিতে হয় এবং এটি ফিক্সড সাইজ তাই রানটাইমে আমরা এটার সাইজ বাড়াতে পারিনা । অবশ্য এই সমস্যা দূর করতে জাভাতে কালেকশন ফ্রেমওয়ার্ক ব্যবহার করতে পারি যেগুলো পর্যায়ক্রমে আমরা আলোচনা করবো ।

## এক্সপ্রেশান(Expressions), স্টেটমেন্ট(Statements) এবং ব্লক(Blocks)

আমরা ইতিমধ্যে ভেরিয়েবল এবং অপারেটর সম্পর্কে জেনে ফেলেছি, এবার তাহলে আমরা জেনে নিই এক্সপ্রেশান কি।

### এক্সপ্রেশান(Expressions)

এক্সপ্রেশান হচ্ছে কতগুলো ভ্যারিয়েবল, অপারেটর এবং মেথড বা ফাংশান কল এর মাধ্যমে একটি আউটপুট তৈরি করার জন্যে যে কোড লেখা হয়। উদাহরণ-

```
int cadence = 0;
anArray[0] = 100;

int result = 1 + 2;
if (value1 == value2)
    System.out.println("value1 == value2");
```

উপরের `cadence = 0` একটি এক্সপ্রেশান। এটির “=” অপারেটরের মাধ্যমে একটি ভ্যালু `cadence` ভ্যারিয়েবল এ এসাইন হয়। তারপর `anArray[0] = 100` এই এক্সপ্রেশানের মাধ্যমে `anArray` এরে এর প্রথম ঘরে 100 এসাইন করা হল।

`1 + 2` একটি এক্সপ্রেশান যা “+” অপারেটর এর মাধ্যমে দুটি সংখ্যা যোগ হয় এবং “=” অপারেটর এর মাধ্যমে `result` ভ্যারিয়েবল এ এসাইন হয়। সুতরাং এখানে দুইটা এক্সপ্রেশান।

জাভা প্রোগ্রামিং ল্যাংগুয়েজ কম্পাউন্ড এক্সপ্রেশান সাপোর্ট করে। এর মানে হচ্ছে অনেকগুলো ছোট ছোট এক্সপ্রেশান নিয়ে আমরা একটি বড় এক্সপ্রেশান তৈরি করতে পারি। একটি এক্সপ্রেশান মূলত একটি নির্দিষ্ট ডাটাটাইপ এর ভ্যালু প্রদান করে, সুতরাং কম্পাউন্ড এক্সপ্রেশান এর ক্ষেত্রে সব এক্সপ্রেশান এর ফলাফল একি ডাটাটাইপ এর হতে হবে।

```
1 * 2 * 3
```

এখানে `1 * 2` একটি এক্সপ্রেশান যার ইন্টিজার টাইপ এর ডাটাটাইপ এর আউটপুট প্রদান করে, এবং এটি যখন আবার 3 এর মান্লিগাই করা হয়, তখনও এর আউটপুট ইন্টিজার টাইপ হয়।

তবে কম্পাউন্ড এক্সপ্রেশান এর ক্ষেত্রে এন্টিগিউটি দূর করার জন্যে ব্রেস “()” ব্যবহার করা উত্তম। উদাহরণ -

```
x + y / 100
```

এবং  $(x + y) / 100$

এই দুটি এক্সপ্রেশান এর ফলাফল ভিন্ন হবে।

তবে যদি এক্সপ্রেশান এর অর্ডার ব্রেস দিয়ে না ঠিক করে দেওয়া হয় তবে অপারেটর এর অগ্রগণ্যতা(precedence) অনুযায়ী এক্সপ্রেশান এর অর্ডার নির্ধারিত হয়।

### স্টেটমেন্টস(Statements)

স্টেটমেন্টস হচ্ছে অনেকটা একটা পূর্ণাঙ্গ বা সার্থক বাংলা বাক্যের মতো। তবে প্রোগ্রামিং এর ভাষায় এটি হচ্ছে- একটি ছোট ইউনিট অব কোড যা কিনা এক্সিকিউশান করা যায়। কতগুলো এক্সপ্রেশান শেষে সেমিকোলন (;) দিয়ে শেষ করলে স্টেটমেন্ট হয়ে যায়। যেমন-

- এসাইনমেন্ট এক্সপ্রেশান
- ++ অথবা-- এর ব্যবহার
- মেথড/ফাংশান কল
- নতুন অবজেক্ট তৈরি করা, ইত্যাদি।

এদেরকে এক্সপ্রেশানাল স্টেটমেন্ট বলা হয়।

```
// এটি এসাইনমেন্ট স্টেটমেন্ট
aValue = 8933.234;
// এটি ইনক্রিমেন্ট স্টেটমেন্ট
aValue++;
// এখানে একটি মেথড কল করা হয়েছে
System.out.println("Hello World!");
// এখানে একটি অবজেক্ট তৈরি করা হয়েছে
Bicycle myBike = new Bicycle();
```

আরও দু-ধরনের স্টেটমেন্ট আছে- ডিক্লারেশান স্টেটমেন্ট –

```
double aValue = 8933.234;
```

কন্ট্রোল ফ্লো স্টেটমেন্ট – এটি নিয়ে পরবর্তী চ্যাপ্টারে আরও বিস্তারিত বলা হবে।

### ব্লকস(Blocks)

একটি কারলি ব্রেস “{}” এর মাঝে শূণ্য অথবা একাধিক স্টেটমেন্ট থাকলে তাকে ব্লক বলা হয়। উদাহরণ-

```
class BlockDemo {
    public static void main(String[] args) {
        boolean condition = true;
        if (condition) { //এখানে ব্লক -১ এর শুরুর
            System.out.println("Condition is true.");
        } // এখানে ব্লক -১ শেষ
        else { // এখানে ব্লক-২ শুরুর
            System.out.println("Condition is false.");
        } // এখানে ব্লক-২ শেষ
    }
}
```



## # পাঠ ৪: কন্ট্রোল ফ্লো -লুপিং- ব্রাঞ্চিং

- ইফ-দেন-ইলস
- সুইচ
- ফর লুপস
- হুয়াইল লুপ
- ডু-ইয়াইল লুপ
- ব্রেক স্ট্যাটমেন্ট
- কন্টিনিউ স্ট্যাটমেন্ট
- রিটার্ন স্ট্যাটমেন্ট
- সারসংক্ষেপ

আমাদের সোর্সকোডে -এ যেসব স্টেটমেন্ট থাকে তা সাধারণত উপর থেকে নিচে যে অর্ডার এ দেওয়া থাকে সেই অর্ডারেই এক্সিকিউট হয়। কিন্তু কন্ট্রোল ফ্লো স্টেটমেন্ট এই অর্ডারকে ভেঙ্গে বিভিন্ন ডিসিশান মেকিং, লুপিং এবং ব্রাঞ্চিং এর মাধ্যমে একটি নির্দিষ্ট কোড ব্লক-কে এক্সিকিউট করে।

কন্ট্রোল ফ্লো স্টেটমেন্ট গুলি হচ্ছে -

- ডিসিশান-মেকিং স্টেটমেন্ট (if-then, if-then-else, switch)-
- লুপিং স্টেটমেন্ট (for, while, do-while)
- এবং ব্রাঞ্চিং স্টেটমেন্ট (break, continue, return)

`if-then` স্টেটমেন্ট হচ্ছে সব চেয়ে বেসিক কন্ট্রোল ফ্লো স্টেটমেন্ট।

আমরা যদি একটি প্রোগ্রাম এর একটি নির্দিষ্ট কোড ব্লক শুধু মাত্র একটি বিশেষ কন্ডিশান বা শর্ত সাপেক্ষে এক্সিকিউট করতে চাই তাহলে আমরা `if-then` স্টেটমেন্ট ব্যবহার করি-

উদাহরণ-

```
int x = 10;

if( x < 20 ){
    System.out.print("This is if statement");
}
```

উপরের কোড ব্লকটিতে আমরা শুধু মাত্র x এর মান 20 হলেই তা প্রিন্ট করতে চাই।

`if` স্টেটমেন্ট এর পেরেন্থেসিস “()” মাঝে একটি বুলিয়ান এক্সপ্রেশান থাকে। বুলিয়ান এক্সপ্রেশান হচ্ছে এক ধরনের এক্সপ্রেশান যার ফলাফল শুধুমাত্র `true` অথবা `false` হতে পারে। এই বুলিয়ান এক্সপ্রেশানটির মান যদি `true` হয় তাহলে এই if স্টেটমেন্ট এর ব্লকটি এক্সিকিউট হবে, নতুবা হবে না।

তবে আমাদের প্রথম কন্ডিশান বা শর্ত বা বুলিয়ান এক্সপ্রেশান যদি সত্যি না হয়, এবং এক্ষেত্রে আমরা অন্য একটি ব্লক অব কোড এক্সিকিউট করতে চাই, তাহলে `if-then-else` স্টেটমেন্ট ব্যবহার করি। উদাহরণ-

```

if( x < 20 ){
    System.out.print("This is if statement");
}else{
    System.out.print("This is else statement");
}

```

উপরের উদাহরণটি-তে একটি কন্ডিশান বা বুলিয়ান এক্সপ্রেসান ছিল, কিন্তু আমাদের মাঝে মাঝে একাধিক কন্ডিশান থাকতে পারে। তাহলে আরেকটি উদাহরণ দেখা যাক-

```

int score = 76;
char grade;

if (score >= 90) {
    grade = 'A';
} else if (score >= 80) {
    grade = 'B';
} else if (score >= 70) {
    grade = 'C';
} else if (score >= 60) {
    grade = 'D';
} else {
    grade = 'F';
}

System.out.println("Grade = " + grade);

```

উপরের উদাহরণটি যদি আমরা রান করি তাহলে output হবে -

```
Grade = C
```

এখানে প্রথম বুলিয়ান এক্সপ্রেসানটি যদি `true` হয়, তাহলে `grade = 'A';` কোড ব্লকটি এক্সিকিউট হবে, আর `true` না হয়, তাহলে পরের কোড ব্লক, অর্থাৎ `else if (score >= 80)` এক্সপ্রেসানটি ইভালুয়েট করা হবে, এবং এটি যদি `true` হয় তাহলে এর কার্লি ব্রেস `{}` এর মাঝের কোড ব্লকটি এক্সিকিউট হবে। অর্থাৎ আমাদের যদি অনেকগুলো কন্ডিশান থাকে তাহলে আমরা `if` কন্ডিশান এর সাথে `else if` দিয়ে সেগুলো-কে এড করতে পারি। এই কন্ডিশান গুলোর মধ্যে যে কোন একটি এক্সপ্রেসান যদি `true` হয় তাহলে সেই ব্লক এর কোডটি এক্সিকিউট হবে।

এখানে লক্ষ্য রাখতে হবে যে, প্রথম এক্সপ্রেসানটি যদি `true` হয়, তাহলে কিন্তু বাকি কন্ডিশান গুলো আর ইভালুয়েট হবে না। অর্থাৎ রান টাইমে এই কোড ব্লক গুলো একদম প্রথম `if` কন্ডিশান থেকে যতক্ষণ পর্যন্ত কোন `true` এক্সপ্রেসান না পাওয়া যায়, ঠিক ততক্ষণ পর্যন্ত এক্সপ্রেসান গুলো ইভালুয়েট হবে। আমাদের উদাহরণটিতে - প্রথম, দ্বিতীয় এবং তৃতীয় এই তিনটি এক্সপ্রেসান ইভালুয়েটেড হয়েছে, এবং তৃতীয়টিতে `true` এক্সপ্রেসান পাওয়া গেছে, এবং `grade = 'C';` এই কোড ব্লকটি এক্সিকিউট হয়েছে।

এভাবে আমাদের যদি একাধিক কন্ডিশান এর জন্য আমরা `if-then-else` ব্যবহার করে কোড লিখতে পারি। যদি একাধিক

## Switch

আমাদের কোড এ যদি একাধিক এক্সিকিউশান পথ থাকে তাহলে, আমরা `if-then` এবং `if-then-else` ব্যবহার করে কোড লিখতে পারি। তবে এর পরিবর্তে `switch` স্ট্যাটমেন্ট ও ব্যবহার করতে পারি। উদাহরণ-

```
public static String getMonth(int month) {
    String monthString;

    switch (month) {
        case 1:
            monthString = "January";
            break;
        case 2:
            monthString = "February";
            break;
        case 3:
            monthString = "March";
            break;
        case 4:
            monthString = "April";
            break;
        case 5:
            monthString = "May";
            break;
        case 6:
            monthString = "June";
            break;
        case 7:
            monthString = "July";
            break;
        case 8:
            monthString = "August";
            break;
        case 9:
            monthString = "September";
            break;
        case 10:
            monthString = "October";
            break;
        case 11:
            monthString = "November";
            break;
        case 12:
            monthString = "December";
            break;
        default:
            monthString = "Invalid month";
            break;
    }
    return monthString;
}
```

## For Loop



যখন আমাদের একই কাজ বারবার করার প্রয়োজন হয় তখন আমরা লুপ ব্যবহার করি । ধরুন আপনাকে ১০ বার বাংলাদেশ শব্দটি প্রিন্ট দিতে বলা হল তাহলে আপনি `System.out.println("Bangladesh");` দশবার না লিখে For Loop ব্যবহার করতে পারেন ।

```
for(int i=1;i<=10;i++)
{
    System.out.println("Bangladesh");
}
```

উপরের কোড টুকুর জন্য Bangladesh শব্দটি ১০ বার প্রিন্ট হবে । আমরা ইচ্ছা করলে Bangladesh শব্দটি অসংখ্যক বার প্রিন্ট দিতে পারি এভাবে-

```
for(;;)
{
    System.out.println("Bangladesh");
}
```

উপরের লুপটাকে infinitive loop বলে ।

For-Each Loop: একটা এরের সবগুলো এলিমেন্টকে এক্সেস করার জন্য আমরা For-Each Loop ব্যবহার করতে পারি । নিচের উদাহরণটি দেখুনঃ

```
public class ForEachLoop {
    public static void main(String[] args) {
        int[] arr={3,6,9,10,30};
        for(int i:arr){
            System.out.println(i);
        }
    }
}
```

উপরের কোডটুকুর জন্য আউটপুট আসবে এমনঃ

```
3
6
9
10
30
```

যতক্ষণ এরের সব এলিমেন্ট প্রিন্ট না হবে ততক্ষণ লুপটি চলবে ।

## While Loop

ফর লুপের মতই যতক্ষণ লুপের কন্ডিশন সত্য হয় ততক্ষণ while loop তার ভিতরের স্টেটমেন্ট এক্সিকিউট করতে থাকে ।

```
while (condition) {
    //block of statements
    statement 1;
    statement 2;
    .....
    statement n;
}
```

উদাহরণঃ

```
class WhileLoopExample {
    public static void main(String args[]){
        int i=7
        while(i>1){
            System.out.println(i);
            i--;
        }
    }
}
```

আউটপুটঃ

```
7
6
5
4
3
2
```

Infinite while loop:

উদাহরণঃ

```
while(1==1){
    System.out.println("Bangladesh");
}
```

উপরের কোডটি দেখে বলুনতো Bangladesh শব্দটি মোট কতবার প্রিন্ট হবে !!!

কারো মনে প্রশ্ন হতে পারে যে for loop দিয়েও তো এসব করা যায় তাহলে আবার while loop কেন? মনে করুন আপনার এক বন্ধু আপনাকে ফোন করে বললো , তুই ১ ঘণ্টার জন্য (for) ওখানে দাঁড়িয়ে থাক আমি আসছি ! আরেকজন ফোন করে বললো আমি যতক্ষণ (while) না আসবো তুই ওখানে ততক্ষণ দাঁড়িয়ে থাক ।

প্রথম ক্ষেত্রে আপনি জানেন যে আপনাকে ঠিক কতক্ষণ দাঁড়িয়ে থাকতে হবে । প্রোগ্রামিং এ যদি আপনি জানেন যে ঠিক কতবার এই কাজটি আমাকে করতে হবে তাহলে সেক্ষেত্রে আপনি for loop ব্যবহার করতে পারেন । ২য় ক্ষেত্রে আপনি ঠিক জানেননা যে কতক্ষণ আপনাকে দাঁড়িয়ে থাকতে হবে । যখন আমরা জানিনা যে ঠিক কতবার লুপ চালাতে হবে সেসব ক্ষেত্রে আমরা while loop ব্যবহার করতে পারি । যেমনঃ আপনি যদি একটি টেক্সট ফাইল থেকে লাইন বাই লাইন ইনপুট নিতে চান তখন আপনি while loop ব্যবহার করতে পারেন কারণ আপনি ঠিক জানেন না যে কতটা লাইন ইনপুট নিলে ফাইলটার শেষ লাইনে পৌঁছানো যাবে ।

## do-while loop

যখন আমরা ঠিক কতবার লুপটি চলবে তা জানিনা কিন্তু মিনিমাম একবার এক্সিকিউট করার দরকার পড়ে তখন do-while loop ব্যবহার করতে পারি ।

```
do {
    // Statements
}while(condition);
```

এক্ষেত্রে কন্ডিশন টেস্ট হবার আগেই স্টেটমেন্ট টি এক্সিকিউট হয় ।

উদাহরনঃ

```
public class DowhileLoop{

    public static void main(String args[]) {
        int i = 5;

        do {
            System.out.print(i);
            i++; //increment by 1
            System.out.print("\n");
        }while( i < 10 );
    }
}
```

আউটপুটঃ

```
5
6
7
8
9
```

চলবে ----



## পাঠ ৫: অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং-১

- অবজেক্ট
- ক্লাস
- ইন্টারফেইস
- অ্যাবস্ট্রাক্ট ক্লাস
- স্ট্যাটিক মেম্বর
- অবজেক্ট ওরিয়েন্টেড কনসেপ্ট
- ইনহেরিট্যান্স
- পলিমরফিজম
- ইনকেপসুলেশন
- সারসংক্ষেপ

শুরুতে বস্তুর ধারণা নিয়ে একটি ছোট্ট ব্যাখ্যা দেই, পরবর্তীতে আমি এই কথাগুলো আরো ব্যাখ্যা করে বলবো। আমরা সবাই কম্পিউটার ব্যবহার করি, যারা একটু বেশি কৌতূহলী তারা নিশ্চয় কম্পিউটারের বক্স খুলে খুলে দেখে ফেলেছে যে, এর মধ্যে নানা রকম যন্ত্রাংশ থাকে, যেমন, র‍্যাম, হার্ডডিস্ক, মাদারবোর্ড, সিপিইউ, কুলিং ফ্যান ইত্যাদি। এইসব মিলেই কম্পিউটার। কিন্তু মজার ব্যাপার হলো, এর সবই একটি কোম্পানি তৈরি করেনি। কেও র‍্যাম তৈরি করে, কেওবা মাদারবোর্ড, কেও বা আবার সিপিইউ। কিন্তু সবাই আলাদা আলাদা ভাবে সবকিছু তৈরি করলেও আমরা যখন পুরো কম্পিউটারটি এসেম্বল করি, কি সুন্দর ভাবে সব ঠিক ঠাক ভাবে লেগে যায়, কোন সমস্যা হয় না। একজন ম্যাকানিক-ও যার নাকি কম্পিউটার সায়েন্স এ ডিগ্রি নেই, সেও জানে কিভাবে সব কিছু এসেম্বল করতে হয়। র‍্যাম এর মধ্যে কি আছে সে সম্পর্কে তার কোনই ধারণা নেই, কিংবা সিপিইউ। অবজেক্ট ওরিয়েন্টেড কনসেপ্ট মূল ব্যাপারটি হলো এটি। একটা সিস্টেমে অনেক গুলো কম্পোনেন্ট থাকতে পারে, কিন্তু সব কম্পোনেন্ট গুলো কেও একা তৈরি করবে না এইটাই স্বাভাবিক, এবং এগুলো এমন ভাবে তৈরি করা হয় যাতে খুব সহজেই এদেরকে এসেম্বল করে পুরো সিস্টেম দাড়া করানো যায়।

অবজেক্ট ওরিয়েন্টেড কনসেপ্ট এর ধারণার সাথে পরিচিত হতে হলে শুরুতে আমাদের কিছু টার্ম বা শব্দের সাথে পরিচিত হতে হয়। আমি শুরুতে এনালজি বা উপমা দিয়ে বুঝানো চেষ্টা করবো, তারপর মূল বিষয়ে চলে আসবো।

### অবজেক্ট (Object):

এর মানে হচ্ছে বস্তু। পৃথিবীতে যা কিছু দেখি, অনুভব করি, তার সবই বস্তু। যেমন- মোবাইল ফোন, চশমা, এমনকি আমি নিজেও একটি অবজেক্ট। আমরা যেহেতু প্রোগ্রামার, এখন একটু সেভাবে কথা বলি। প্রোগ্রামিং এ একটা ধারণাও অবজেক্ট। অবজেক্ট কে কিভাবে দেখা হচ্ছে তা নির্ভর করে যে দেখছে তার উপর। মনে করা যাক, একটি অফিসের বড়ো কর্তা (CEO) সে দেখবে, এমপ্লয়ি, বিল্ডিং, ডিভিশন, নোটপত্র, বেনিফিট প্যাকেজ, লাভ ক্ষতির হিসেব এগুলো অবজেক্ট। একজন আর্কিটেক্ট দেখবে, তার প্ল্যান, মডেল, এলভেশান, ডোনেজ, ডেস্টিল, আর্মাচার ইত্যাদি। সেভাবে একজন সফটওয়্যার ইঞ্জিনিয়ারের অবজেক্ট হলো, স্ট্যাক, কিউ, উইন্ডো, চেক বক্স ইত্যাদি। অবজেক্ট এর একটি স্টেট থাকে। স্টেট হলো কিছু তথ্য যা দিয়ে ওই অবজেক্টকে আলাদা করা যায়, এবং তার বর্তমান অবস্থান জানা যায়। যেমন একটি ব্যাংক একাউন্ট স্টেট হতে পারে কারেন্ট ব্যালেন্স। একটা অবজেক্ট এর মধ্যে আরেকটি অবজেক্ট থাকতে পারে, যা ওই অবজেক্ট এর স্টেট হতে পারে।

অবজেক্ট সাধারণত কিছু কাজ করে থাকে যাকে বলে তার বিহেভিয়ার। যেমন ধরা যাক, সাইক্যালের চাকা, চাকার স্টেট হতে পারে এর ব্যাসার্ধ, পরিধি, গতি ইত্যাদি এবং চাকার বিহেভিয়ার হলো এটি ঘুরে। এখন যেহেতু আমরা সাইক্যাল এর চাকাকে কে আমরা প্রোগ্রামিং এর মাধ্যমে প্রকাশ করবো, সুতরাং এগুলোকে আমরা ডায়ারিয়েবল এ রাখবো। আর বিহেভিয়ার গুলোকে আমরা ফাংশন এর মাধ্যমে লিখি। আমরা এর আগে যাকে ফাংশন বলে এসেছি এখন থেকে আমরা ফাংশন কে ফাংশন বলবো না, আমরা এদেরকে মেথড বলবো।

সুতরাং আমরা জানলাম, অবজেক্ট এর দুইটা জিনিস থাকে, স্টেট ( অর্থাৎ নিজের সম্পর্কে ধারণা) এবং মেথড (সে কি কি কাজ করতে পারে)।

### ক্লাস(Class)

অবজেক্ট সম্পর্কে আমরা জানলাম। ক্লাস হলো সেই অবজেক্টটি তৈরি করার প্রক্রিয়ার একটি অংশ। মনে করি আমরা একটি কলম বানাতে চাই, শুরুতে আমরা কোন রকম চিন্তা ভাবনা না করে ফু দিয়ে একটা কিছু বানিয়ে ফেলতে পারি না। আমরা এর জন্যে পরিকল্পনা করি- কলমাটা দেখতে কেমন হবে, এটি লম্বা কতটুকু হবে, কলমটি কি কি কাজ করবে ইত্যাদি। এই পরিকল্পনা গুলো আমরা আমরা কোথাও লিখে রাখি। আমাদের এই লেখা ডকুমেন্টটি আসলে ক্লাস। সহজ একটি ব্যাপার।

অবজেক্ট ওরিয়েন্ট কনসেপ্ট তিনটি ধারণার উপর প্রতিষ্ঠিত।

এক, ইনহেরিট্যান্স- নাম থেকেই বুঝা যাচ্ছে যে এখানে বংশগতির একটা বিষয় চলে এসেছে। আসলেও তাই। ধরা যাক একটি একটা চাকা। নানা রকম চাকা হতে পারে, যেমন বাসের চাকা, সাইক্যাল এর চাকা, মটর সাইক্যাল এর চাকা ইত্যাদি। সব চাকার মধ্যেই কিন্তু কিছু কমন ব্যাপার আছে, এটির ব্যাসার্ধ আছে, পরিধি আছে এবং এটি ঘুরে। সুতরাং আমরা একটা চাকা নামের অবজেক্ট বানাতে পারি যা বাকি সব চাকা(বাস, সাইক্যাল এর) পূর্বপুরুষ। এতে আমাদের বেশি কিছু সুবিধা আছে, প্রধান সুবিধে হলো, কমন জিনিস গুলো নিয়ে আমাদের পূর্বপুরুষ তৈরি করার কাজ একবার করে ফেললেই হচ্ছে, উত্তরপুরুষ গুলোতে আপনা আপনি সেই বৈশিষ্ট্যগুলো চলে আসবে।

দুই, এনক্যাপসুলেশন- মানে হলো জিনিসপত্র ক্যাপসুলের মধ্যে ভরে রাখা। আমরা অনেকেই ক্যাপসুল মেডিসিন খেয়েছি, এটির বাইরে একটা আবরণ দিয়ে সব কিছু ভেতরে আটকানো থাকে। ব্যাপারটি এমনি।

তিন, পলিমরফিজম – বহুরূপিতা। অর্থাৎ একি অঙ্গে নানা রূপ। একটা অবজেক্ট নানা সময় নানা রকম রূপ ধারণ করতে পারে।

তবে কেন এই অবজেক্ট ওরিয়েন্টেড কনসেপ্ট লাগবে সেটি নিয়ে প্রশ্ন হতে পারে। এবার তাহলে এই প্রশ্নের উত্তর ব্যাখ্যা করা যাক।

আমাদের পরিচিত প্রথাগত বা প্রসিডিউরাল প্রোগ্রামিং ল্যাংগুয়েজ যেমন- সি এর কিছু অসুবিধা রয়েছে। আমরা চাইলেই সহজে পুনরায় ব্যবহারযোগ্য কম্পোনেন্ট তৈরি করতে পারি না। সবচেয়ে বড় অসুবিধা হলো আমরা চাইলেই একটি প্রোগ্রাম থেকে একটি ফাংশন কপি করে অন্য একটি প্রোগ্রামে ব্যবহার করতে পারি না কারণ ফাংশন গুলো সাধারণত কিছু গ্লোবাল ডেরিয়েবল এবং অন্যান্য ফাংশন এর উপর নির্ভর করে। এই ল্যাংগুয়েজ গুলো হাই-লেভেল এবস্ট্রাকশন এর জন্যে মানানসই নয়। যেমন সি যে কম্পোনেন্ট গুলো ব্যবহার করে সেগুলো খুব লো-লেভেল-এর যা দিয়ে একটি বাস্তব জগতের সমস্যাকে খুব সহজে চিত্রায়ণ (portray) করা সম্ভব হয় না। কাস্টমার রিলেশনশিপ ম্যানেজমেন্ট বা সিআরএম অথবা ফুটবল খেলাকে সহজে সি দিয়ে চিত্রায়ণ করা কঠিন।

১৯৭০ সালের যুক্তরাষ্ট্রের প্রতিরক্ষা অধিদপ্তরের একটি টাস্কফোর্স তদন্ত করে বের করার চেষ্টা করে কেন আইটি(IT) বাজেট সবসময় নিয়ন্ত্রণ করা সম্ভব হয় না। সেগুলোর মধ্যে প্রধান গুলো এমন- ৮০% বাজেট শুধুমাত্র সফটওয়্যার এর জন্যে ব্যয় হয় আর বাকি ২০% ব্যয় হয় হার্ডওয়্যার এর জন্যে। এর মধ্যে ৮০% ব্যয় হয় শুধুমাত্র সফটওয়্যার মেইনটেইন করার জন্যে, বাকি ২০% তৈরি হয় সফটওয়্যার তৈরি করার জন্যে। হার্ডওয়্যার গুলো সহজেই রিইউজ বা পুনরায় ব্যবহার করা যায় এবং এতে এদের ইন্টিগ্রিটি নষ্ট হয় না, এবং একটি হার্ডওয়্যার একটি বিশেষ অংশ নষ্ট হয়ে গেলে তা সহজেই আলাদা করে ফেলা যায় এবং নতুন একটি দিয়ে রিপ্লেস করা যায়। কিন্তু সফটওয়্যার এর ক্ষেত্রে এমন সম্ভব হয় না, একটি প্রোগ্রাম এর সমস্যার জন্যে অন্য প্রোগ্রাম এর সমস্যা তৈরি হয় ইত্যাদি।

এই সমস্যা সমাধান করার জন্যে এই টাস্কফোর্স পরিশেষে প্রস্তাব করে যে সফটওয়্যার-ও হার্ডওয়্যার এর মতো হওয়া উচিত। পরবর্তীতে তারা তাদের সিস্টেম এর ৪৫০ টি প্রোগ্রামিং ল্যাংগুয়েজ রিপ্লেস করে এডা (Ada) নামে একটি অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং ল্যাংগুয়েজ ব্যবহার করে।

**চলবে -----**

## ## ইনহেরিট্যান্স-

এবার আমরা অবজেক্ট ওরিয়েন্টেড কনসেপ্ট-এর আরও ভেতরে প্রবেশ করবো। শুরুতেই আমরা ইনহেরিটেন্স নিয়ে আলোচনা করি।

ইনহেরিটেন্স নিয়ে কথা বলতে হলে এর সাথে আরেকটি বিষয় চলে আসে সেটি হলো অবজেক্ট কম্পোজিশান। এটি মোটামুটিভাবে একটু কঠিন অন্যান্য টপিক থেকে। তাই এই টপিকটি পড়ার সময় একটু ধৈর্য নিয়ে পড়তে হবে।

তো শুরু করার যাক-

প্রথমেই আমরা কথা বলবো Is - A এবং Has - A নিয়ে।

যেহেতু আমরা জাভা প্রোগ্রামিং শুরু করেছি, তাই আমরা যতই এর ভেতরে প্রবেশ করতে শুরু করবো, ততই বুঝতে শুরু করবো যে ক্লাস আসলে একটা স্ট্যান্ড এলোন কম্পোনেন্ট নয়, বরং এটি অন্যান্য ক্লাসের উপর নির্ভর করে। অর্থাৎ ক্লাস গুলো একটি রিলেশন মেইনটেইন করে চলে। এই রিলেশন গুলো সাধারণত দুই ধরনের হয়- Is - A এবং Has - A।

আমাদের বাস্তব জগৎ থেকে একটা এনালজি দেয়া যাক। যেমন একটি বিড়াল, কিংবা কার অথবা বাস। বিড়াল হচ্ছে একটি প্রাণি। কার এর থাকে চাকা এবং ইঞ্জিন। বাস এরও থাকে চাকা এবং ইঞ্জিন। আবার কার এবং বাস দুটিই ভেহিকল বা যান।

এখানে যে উদাহরণ গুলো দেয়া হয়েছে এর সবগুলো মূলত Is - A অথবা Has - A রিলেশনশিপ মেইনটেইন করে। যেমন -

A cat is an Animal (বিড়াল একটি প্রাণি) A car has wheels (কার এর চাকা আছে।)

A car has an engine (কার এর একটি ইঞ্জিন আছে।)

তো ব্যপারটি একদম সহজ। ঠিক এই ব্যপারটিকে আমরা আমাদের অবজেক্ট ওরিয়েন্টেড কনসেপ্ট এর মাধ্যমে বলতে পারি। যখন কোন অবজেক্ট এর মাঝে Is - A এই সম্পর্কটি দেখবো তাকে বলবো ইনহেরিটেন্স। আবার যখন কোন অবজেক্ট এর মাঝে Has - A এই সম্পর্কটি দেখবো তখন সেই ব্যপারটিকে বলবো অবজেক্ট কম্পোজিশান।

ইনহেরিটেন্স মূলত একটি ট্রি-রিলেশনশিপ। অর্থাৎ এটি একটি অবজেক্ট থেকে ইনহেরিট করে আসে।

আর যখন আমরা অনেকগুলো অবজেক্ট নিয়ে আরেকটি অবজেক্ট তৈরি করবো তখন সেই নতুন অবজেক্ট হলো মেইড-আপ বা নতুন তৈরি করা অবজেক্ট এই ঘটনাটি হলো কম্পোজিশান।

এর সবই আসলে একটি কনসেপ্ট এবং আইডিয়া থেকে এসেছে, সেটি হলো কোড রিইউজ করা এবং সিম্পল করা। যেমন দুটি অবজেক্ট এর কোড এর কিছু অংশ যদি কমন থাকে তাহলে আমরা সেই অংশটিকে দুইটি ক্লাসের মধ্যে পুনরায় না লিখে বরং তাকে ব্যবহার করতে পারি।

ধরা যাক, আমরা দুটি অবজেক্ট তৈরি করতে চাই- Animal এবং Cat

আমরা জানি যে সব Animal খায়, ঘুমায়। সুতরাং আমরা এই ক্লাসে এই দুটি বৈশিষ্ট্য আমরা এই ক্লাসে লিখতে পারি। আবার যেহেতু আমরা জানি যে Cat হচ্ছে একটি Animal। সুতরাং আমরা যদি এমন ভাবে কোড লিখতে পারি, যাতে করে এই Cat ক্লাসের মধ্যে নতুন করে আর সেই দুটি বৈশিষ্ট্যের কোড আর লিখতে হচ্ছে না, বরং আমরা এই Animal ক্লাসটিকে রিইউজ করলাম, তাহলে যে ঘটনাটি ঘটবে তাকেই মূলত ইনহেরিটেন্স বলা হয়।



এইভাবে আমরা আরও অন্যান্য Animal যেমন, Dog, Cow ইত্যাদি ক্লাস লিখতে পারি।

কম্পোজিশান তুলনামূলক ভাবে একটু সহজ।

যেমন আমরা একটি Car তৈরি করতে চাই। Car তৈরি করতে হলে আমাদের লাগবে Wheel এবং Engine. সুতরাং আমরা Wheel এবং Engine এই দুটি ক্লাসকে নিয়ে নতুন আরেকটি ক্লাস লিখবো।

এবার তাহলে একটি উদাহরণ দেখা যাক।

```
public class Bicycle {  
  
    // the Bicycle class has three fields  
    public int cadence;  
    public int gear;  
    public int speed;  
  
    // the Bicycle class has one constructor  
    public Bicycle(int startCadence, int startSpeed, int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
  
    // the Bicycle class has four methods  
    public void setCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    public void setGear(int newValue) {  
        gear = newValue;  
    }  
  
    public void applyBrake(int decrement) {  
        speed -= decrement;  
    }  
  
    public void speedUp(int increment) {  
        speed += increment;  
    }  
}
```

উপরের Bicycle ক্লাসটিতে তিনটি ফিল্ড এবং চারটি মেথড আছে। এবার এই Bicycle থেকে আমরা এর একটি সাব-ক্লাস লিখবো-

```

public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
                        int startCadence,
                        int startSpeed,
                        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}

```

এই MountainBike ক্লাসটি উপরে Bicycle এর সব ফিল্ড এবং মেথড গুলো ইনহেরিট করে এবং এতে নতুন করে শুধু একটি ফিল্ড এবং একটি মেথড লেখা হয়েছে। তাহলে আমাদের MountainBike ক্লাসটিতে Bicycle ক্লাসটির সব প্রোপার্টি এবং মেথড অটোম্যাটিক্যালি পেয়ে গেলো।

এখানে এ Bicycle হচ্ছে সুপার ক্লাস(Super Class) এবং MountainBike হচ্ছে সাব-ক্লাস(Sub Class)। অর্থাৎ যে ক্লাস থেকে ইনহেরিট করা হয় তাকে বলা হয় সুপার ক্লাস এবং যে ক্লাস সাব ক্লাস থেকে ইনহেরিট করে

### মেথড অভাররাইডিং(Method Overriding)

যদিও সাব-ক্লাস সুপার-ক্লাসের সব গুলো প্রোপার্টি এবং মেথড ইনহেরিট করে, তবে সাব-ক্লাসে সুপার ক্লাসের যে কোন প্রোপার্টি বা মেথড কে অভাররাইড করা যায়।

একটি উদাহরণ দেখা যাক-

```

public class Circle {
    double radius;
    String color;

    public Circle(double radius, String color) {
        this.radius = radius;
        this.color = color;
    }

    public Circle() {
        radius = 1.0;
        color = "RED";
    }

    public double getArea() {
        return radius * radius * Math.PI;
    }
}

```

এই ক্লাসটিতে `getArea()` মেথড একটি বৃত্তের ক্ষেত্রফল রিটার্ন করে।

এখন আমরা এই ক্লাসটিকে এক্সটেন্ড ( `extends` ) করে নতুন আরেকটি ক্লাস লিখবো-

```

public class Cylinder extends Circle {
    double height;

    public Cylinder() {
        this.height = 1.0;
    }

    public Cylinder(double radius, String color, double height) {
        super(radius, color);
        this.height = height;
    }

    @Override
    public double getArea() {
        return 2 * Math.PI * radius * height + 2 * super.getArea();
    }
}

```

এই ক্লাসটিতে `Circle` এর মেথডটি আমরা সাধারণ ভাবেই পেয়ে যাবো। `Cylinder` এর ক্ষেত্রফল নির্ধারণ করতে হলে `getArea()` কল করলেই হয়ে যাচ্ছে। কিন্তু আমরা জানি যে `Circle` এবং `Cylinder` এর ক্ষেত্রফল একভাবে নির্ধারণ করা যায় না। এক্ষেত্রে আমরা যদি `Circle` এর মেথডটি কে ব্যবহার করি তাহলে আমাদের ক্ষেত্রফলের মান ভুল আসবে। এই সমস্যা সমাধান করার জন্যে আমরা আমাদের `Cylinder` ক্লাসটিতে `getArea()` মেথডটিকে পুনরায় লিখেছি।

এখানে লক্ষ্য রাখতে হবে যে, দুটি মেথড এর সিগনেচার, রিটার্ন-টাইপ এবং প্যারামিটার লিস্ট একই রকম হতে হবে।

এখন আমরা যদি `Cylinder` ক্লাস-এর `getArea()` মেথড কল করি, তাহলে অভাররাইডেড মেথডটি কল হবে।

### অ্যানোটেশন(Annotation) `@Override`

`@Override` এই অ্যানোটেশনটি জাভা 1.5 ভার্সনে প্রথম নিয়ে আসা হয়। কোন মেথডকে যদি আমরা অভাররাইড করি তাহলে সেই মেথড এর উপরে `@Override` দেয়া হয়। এটি কম্পাইলারকে ইনফর্ম করে যে, এই মেথডটি সুপার ক্লাসের অভাররাইডেড মেথড।

তবে এটি অপশনাল হলেও অবশ্যই ভাল যদি ব্যবহার করা হয়।

### `super` কিওয়ার্ড

আমরা যদি সাব ক্লাস থেকে সুপার ক্লাসের কোন মেথড বা ভেরিয়েবল একসেস করতে চাই তাহলে আমরা এই কিওয়ার্ডটি ব্যবহার করি। কিন্তু আমরা জানি যে সাব ক্লাসে অটোমেটিক্যালি সুপার ক্লাসের সব প্রোপারটি চলে আসে তাহলে এর প্রয়োজনীয়তা নিয়ে প্রশ্ন হতে পারে।

আমরা আবার উপরের `Cylinder` ক্লাসটি আবার দেখি। এই ক্লাসটিতে আমরা নতুন আরেকটি মেথড লিখতে চাই।

```
public double getVolume() {
    return getArea() * height;
}
```

এই মেথডটি-তে `getArea() * height` এই স্ট্যাটমেন্টটি লক্ষ করি। এখানে `getArea()` এই মেথডটি কল করা হয়েছে। আমাদের এই `Cylinder` ক্লাসটিতে `getArea()` মেথডটিকে আমরা `Circle` ক্লাস এর `getArea()` মেথড-কে অভাররাইড করেছি। সুতরাং আমরা যখন এই `Cylinder` ক্লাস থেকে `getArea()` মেথডটি কল করবো তখন আসলে `Cylinder` ক্লাস এর মেথডটি কল হবে।

কিন্তু এক্ষেত্রে আমাদের একটি সমস্যা হচ্ছে যে - আমরা জানি সিলিন্ডারের আয়তন

```
V = Pi * r^2 * h
    = (Pi * r^2) * h
    = Area of Circle * h
```

সুতরাং `Cylinder` ক্লাসের `getArea()` মেথডটি ব্যবহার করা যাচ্ছে না। কারণ সিলিন্ডারের ক্ষেত্রফল-

$$A = (2 * \pi * r * h) + (2 * \pi * r^2)$$

কিন্তু আমরা যদি `Circle` ক্লাস এর মেথডটি ব্যবহার করি তাহলে আমাদের সমস্যা সমাধান হয়ে যায়। এখন যদি আমরা সুপার ক্লাস(`Circle`) এর মেথডটি কল করে এই মেথডটি লিখতে চাই তাহলে -

```
public double getVolume() {
    return super.getArea() * height;
}
```

অর্থাৎ সাব ক্লাসে যদি মেথড অডাররাইড করা হয় এবং তারপরেও কোন কারণে যদি আমাদের সুপার ক্লাসের মেথড কে কল করার প্রয়োজন হয় তাহলে আমরা সুপার( `super` ) কিওয়ার্ডটি ব্যবহার করি।

ইনহেরিটেন্স এর ক্ষেত্রে মনে রাখতে হবে –

জাভা মাল্টিপল ইনহেরিটেন্স সাপোর্ট করে না। এর মানে হচ্ছে আমার একটি ক্লাস শুধুমাত্র একটি ক্লাসকেই ইনহেরিট করতে পারে।

### কনস্ট্রাক্টর অডাররাইডিং(Constructor Overriding)

মেথডের মত কনস্ট্রাক্টরও অডাররাইড এবং ওভারলোড (Overload) করা যায়। অডাররাইড করা যায় বলতে অনেক ক্ষেত্রে কনস্ট্রাক্টর অডাররাইড ম্যান্ডেটরি। প্যারেন্ট ক্যালে যদি এমন কোন কনস্ট্রাক্টর থাকে যেটি প্রাইভেট নয় এবং যেটি এক বা একাধিক ইনপুট প্যারামিটার নিয়ে থাকে তবে চাইল্ড ক্লাসে অবশ্যই সেই কনস্ট্রাক্টর অবশ্যই অডাররাইড করতে হবে। এটা বাধ্যতামূলক। একটা ছোট উদাহরন দিয়ে বিষয়টি আমরা পরিষ্কার করে নিতে পারি।

```
class Animal {

    String animalName ;
    String animalColor ;

    public Animal(String animalName, String animalColor){

        this.animalName = animalName;
        this.animalColor = animalColor;
    }

    public void showName(){
        System.out.println("Animal Name is: "+this.animalName);
    }
    public void showColor(){
        System.out.println("Animal Color is: "+this.animalColor);
    }
}

class Cow extends Animal{

    private String work ;
    public Cow(String animalName, String animalColor) {
//        this.work = "No Work";//This is not valid
        super(animalName, animalColor);//super in constructor have to be on top
        this.work = "Gives Milk";//This is valid
    }

    @Override
    public void showColor() {
        System.out.println("Before showColor in child");
        super.showColor();
    }

    @Override
    public void showName() {
```

```

        super.showName();
        System.out.println("After showName in child");
    }
    public void showDescription(){
        this.showName();
        System.out.println("Animal Work is: "+this.work);
        this.showColor();
    }
}

public class Main {

    public static void main(String[] args) {

        Cow cow = new Cow("White Cow", "White");
        cow.showDescription();
    }
}

```

উপরের কোডটি মন দিয়ে লক্ষ করুন। প্যারেন্ট ক্লাস `Animal` এর মাঝে একটি পাবলিক কনস্ট্রাক্ট আছে যেটি দুটি প্যারামিটার নিয়ে থাকে। তাই এটার চাইল্ড ক্লাসেও আমাদের অবশ্যই একটি কনস্ট্রাক্টর থাকতে হবে যেটির মাঝে প্যারেন্ট ক্লাসের ওই কনস্ট্রাক্টর ইনডোক করতে হবে। এটা ম্যান্ডেটরি। এটি না করলে কোড কম্পাইলেশন এরর শো করবে এবং কম্পাইলই হবেনা। আরো একটি বিষয় চাইল্ড ক্লাসের কনস্ট্রাক্টরের মাঝ থেকে প্যারেন্ট ক্লাসের কনস্ট্রাক্টরকে `super` কিয়াওর্ড দিয়ে কল করতে হবে তবে, `super` কিয়াওর্ড অবশ্যই সবার উপর থাকতে হবে। এমনকি একটি প্রিন্ট স্টেটমেন্টও থাকতে পারবে না। `super` এর পর যা খুশি থাকতে পারে কোন সমস্যা নাই। এছাড়া অন্য একাধিক কনস্ট্রাক্টর ডিক্লেয়ার করার প্রয়োজন হলে সেটাও করতে পারবেন, এটাকে বলা হবে কনস্ট্রাক্টর ওভারলোডিং। যথারীতি এর মাঝেও `super` বাবাজি অধিপত্য বিরাজ করে বসে থাকবে।

তবে মেথড আর কনস্ট্রাক্টরের ওভাররাইডিং এর মাঝে এটা বড় একটা পার্থক্য যে মেথডের ক্ষেত্রে সুপার আপনারা কাজের সুবিধার জন্য যেকোন যায়গার ব্যাবহার করতে পারবেন। তবে কনস্ট্রাক্টরের ক্ষেত্রে আসিন খুবই কঠিন।

### অভারলোডিং অফ মেথড & কনস্ট্রাক্টর ( Overloading of method and constructor ):

অভারলোডিং বলতে খুব সাধারণ ভাষায় বোঝায় একই নামের এবং একই রিটার্ন টাইপের ( নাও হতে পারে ) একাধিক মেথড বা কনস্ট্রাক্টর ( এক্ষেত্রে কোন রিটার্ন টাইপ থাকবে না ) বিদ্যমান থাকা। তার অর্থ দাড়ালো একই ক্লাসের মাঝে একই নামের এবং রিটার্ন টাইপের একাধিক মেথড বা কনস্ট্রাক্টর বিদ্যমান থাকবে। বিষয়টা একটু গোলমালে মনে হচ্ছে তাইনা? আসলে তেমন কিছুই নয়, বরং বিষয়টি অন্য অনেক কিছুর থেকেও অনেক বেশি সরল। একই নামের মেথড বা কনস্ট্রাক্টর থাকলেও তাদের ইনপুট প্যারামিটার কিন্তু একই হবেনা। ইনপুট টাইপ হয়ত ভিন্ন টাইপের হবে নাহয় একটি মেথড থেকে অন্য মেথডের ইনপুট প্যারামিটার সংখ্যা ভিন্ন হবে। উদাহরন সহকারে আমরা আমাদের কনফিউশন দূর করতে পারি। চলুন একটি উদাহরন দেখে নেওয়া যাকঃ

```

public class Main {

    private int initialNumber;
    private int terminalNumber;

    public Main(int initialNumber, int terminalNumber) {

        this.initialNumber = initialNumber;
        this.terminalNumber = terminalNumber;
    }

    public Main(int terminalNumber) {

        this(0, terminalNumber);
    }

    public Main() {

        this(0, 100);
    }

    public void showNumbers() {

        System.out.println("First Number: " + this.initialNumber + ", Second Number: " +
    }

    public static void main(String[] args) {

        Main m = new Main(1, 5);
        m.showNumbers();
        Main m2 = new Main(5);
        m2.showNumbers();
        Main m3 = new Main();
        m3.showNumbers();
    }
}

```

উপরের কোড সেগমেন্টটিতে আরা দেখতে পারছি যে একই `Main` ক্লাসে একই নামের কনস্ট্রাক্টর ৩ টি। খেয়াল করলে দেখা যাবে যে ৩ টি কনস্ট্রাক্টর প্রায় একই কাজ করলেও তাদের ইনপুট প্যারামিটার কিন্তু একই নয়। একেক জন একেক রকম ইনপুট নিয়ে কাজ করছে। এভাবে একই ক্লাসের মাঝে একাধিক কাজের জন্য একাধিক কনস্ট্রাক্টর ব্যবহার করাকে বলা হয় কনস্ট্রাক্টর অভারলোডিং। যেখানে একই কনস্ট্রাক্টরের লোড অভার হয়ে গিয়েছে :P

ওকে, এবার আসা যাক মেথড অভারলোডিং বিষয়ে। কনস্ট্রাক্টরের মত মেথড অভারলোডিংও সেম ম্যাকানিজম ফলো করে। একটি উদাহরণ দিলেই বিষয়টি পরিষ্কার হয়ে যাবেঃ

```

public class Main {

    private int sum(int a, int b){
        return a+b;
    }
    private int sum(int a, int b, int c){
        return a+b+c;
    }
    private int sum(int ... a){

        int result = 0;
        for(int x : a){
            result+=x;
        }
        return result;
    }

    public static void main(String[] args) {

        Main m = new Main();
        System.out.println(m.sum(3, 5));
        System.out.println(m.sum(3, 5, 7));
        System.out.println(m.sum(3, 5, 7, 17));
        System.out.println(m.sum(3, 5, 7, 17, 23));
    }
}

```

উপরোক্ত কোডটিতে দেখুন `sum` মেথডটি ৩ বার লেখা হয়েছে। মেথডের আইডেন্টিফায়ার, রিটার্ন টাইপ সবই ঠিক আছে তবু কাজ করছে! হ্যাঁ কারন আপনার ইনপুট প্যারামিটার ভিন্ন দিয়েছি। প্রথম `sum` মেথড কেবল ২ টি নাম্বারের যোগ করে দিতে পারে। দ্বিতীয়টি পারে ৩ টি নাম্বারের, আর শেষেরটি পারে যত সংখ্যক ইন্টিজার নাম্বারই দেওয়া হোক না কেন সে যোগ করে রেজাল্ট দিবে। সিম্পলি এটাকেই বলা হয় মেথড ওভারলোডিং। যেখানে একই নামের একাধিক মেথড থাকে যাদের নাম এক হলেও ইনপুট প্যারামিটার বা কাজের ধরন সম্পূর্ণ আলাদা হয়।

### অ্যাবস্ট্রাক্ট ক্লাস ( **Abstract Class** ):

অ্যাবস্ট্রাক্ট ক্লাস হল বিশেষ এক ধরনের ক্লাস যেটির মাঝে কমপক্ষে একটি অ্যাবস্ট্রাক্ট মেথড থাকবে। তাহলে প্রশ্ন হল অ্যাবস্ট্রাক্ট মেথড আসলে কি জিনিস। অ্যাবস্ট্রাক্ট মেথড হল এমন এক ধরনের মেথড যেটার কোন বডি নেই। সোজা কথায় যে মেথডের অ্যাক্সেস মডিফায়ার আছে, রিটার্ন টাইপ আছে, মেথডের নাম আছে, ইনপুট প্যারামিটার আছে কিন্তু কোন বডি ডিফাইন করা নেই। বডি এম্পটি বা অ্যাবস্ট্রাক্ট সেজন্য এই মেথডকে অ্যাবস্ট্রাক্ট মেথড বলা হয়েছে। চলুন ছোট্ট একটা উদাহরন দেখে পরে ব্যাখ্যার দিকে যাই।

```

public abstract class Animal{

    public abstract String color();
    public void name(){
        System.out.println("Tiger");
    }
}

```



উপরোক্ত কোডটিতে `Animal` একটি অ্যাবস্ট্রাক্ট ক্লাস। অ্যাবস্ট্রাক্ট ডিক্লেয়ার করার সময় `class` কিওয়ার্ডের আগে `abstract` কিওয়ার্ডটি লিখতে হবে। অ্যাবস্ট্রাক্ট ক্লাস পাবলিক বা ডিফল্ট যেকোনটিই হতে পারে। এই ক্লাস অ্যাবস্ট্রাক্ট ডিক্লেয়ার করার কারন এর মাঝে আমরা একটি অ্যাবস্ট্রাক্ট মেথড ডিক্লেয়ার করেছি যেটির নাম `color`। সহজেই আমরা বুঝতে পারছি যে অ্যাবস্ট্রাক্ট মেথড ডিক্লেয়ার করতে গেলে তার আগে `abstract` কিওয়ার্ডটি ব্যবহার করতে হবে। লক্ষ করে দেখুন `color` মেথডের অ্যাক্সেস মডিফায়ার, রিটার্ন টাইপ (ইনপুট প্যারামিটার দিলে দেওয়া সম্ভব) সবই আছে কিন্তু কোন বডি নেই। এজন্য এই মেথডকে বলা হয়েছে অ্যাবস্ট্রাক্ট মেথড। একটি অ্যাবস্ট্রাক্ট ক্লাসে যেমন একাধিক অ্যাবস্ট্রাক্ট মেথড থাকতে পারে তেমন সাধারণ মেথডও থাকতে পারে প্রচুর পরিমাণ। প্রশ্ন হল কেন এই অ্যাবস্ট্রাক্ট মেথড?

অ্যাবস্ট্রাক্ট ক্লাসের কোন ইন্সট্যান্স ক্রিয়েট করা যায়না যতক্ষন না সবগুলো অ্যাবস্ট্রাক্ট মেথডকে ওভাররাইড করা হচ্ছে। অ্যাবস্ট্রাক্ট মেথড হল একটা রুলের বা নিয়মের মত। এই মেথডের মাধ্যমে বলে দেওয়া হচ্ছে যে, যে ক্লাসই এই `Animal` ক্লাসকে এক্সটেন্ড করবে তাকে অবশ্যই `color` মেথডটি ওভাররাইড করতে হবে এবং নিজস্ব কাজের উপর ভিত্তি করে তাকে। যদি `Animal` ক্লাসকে `Bird` ক্লাস এক্সটেন্ড করে কিন্তু `color` মেথডটি ওভাররাইড না করে তবে `Bird` ক্লাসটিকেও অবশ্যই অ্যাবস্ট্রাক্ট ক্লাস হতে হবে।

```
abstract class Animal{

    abstract void color();
}

abstract class Bird extends Animal{
    //abstract void color(); is present by default
}

class Crow extends Bird{

    @Override
    void color() {
        System.out.println("Black");
    }
}

public class Main {

    public static void main(String[] args) {

        //Animal animal = new Animal(); //This is not possible
        //Bird animal = new Bird(); //This is not possible
        Crow bird = new Crow();
        bird.color();
    }
}
```

উপরোক্ত কোডটিতে আমরা `Bird` ক্লাসের অবজেক্ট কোনভাবেই ক্রিয়েট করতে পারবো না যতক্ষন পর্যন্ত না আমরা এর সব অ্যাবস্ট্রাক্ট মেথড ইমপ্লিমেন্ট করছি। সে কাজটি করা সম্ভব এই ক্লাসটিকে যদি অন্য কোন ক্লাস এক্সটেন্ড করে এবং সব অ্যাবস্ট্রাক্ট মেথড ইমপ্লিমেন্ট করে অথবা অবজেক্ট ক্রিয়েট করার সময় আমরা সব অ্যাবস্ট্রাক্ট মেথড ইমপ্লিমেন্ট করে দেই। প্রথম কাজটি আপনারা পারবেন। এখানে ২য় উপায়টি দেখানো হলঃ

```

public class Main {

    public static void main(String[] args) {

        Bird bird = new Bird() {
            @Override
            void color() {
                System.out.println("White");
            }
        };
        bird.color();
    }
}

```

### ইন্টারফেস ( Interface ):

ইনহেরিট্যান্সের খুব গুরুত্বপূর্ণ এবং কার্যকরী একটি টার্মস হল ইন্টারফেস । ইন্টারফেস ডিক্লেয়ার করতে হয়

`interface` কিওয়ার্ডটি দিয়ে । এটি `public` বা ডিফল্ট যেকোনটিই হতে পারে ক্লাসের মত । আমরা ইতোমধ্যে জেনে ফেলেছি `Abstract` ক্লাস এবং অ্যাবস্ট্রাক্ট মেথড কি জিনিস । অ্যাবস্ট্রাক্ট ক্লাস এবং মেথড বুঝে থাকলে ইন্টারফেস বুঝতে পারা খুব বেশি কঠিন কিছুই নয় । ইন্টারফেস এমন একটি ক্লাস যেখানে সবগুলো মেথডই অ্যাবস্ট্রাক্ট । অর্থাৎ ১০০% অ্যাবস্ট্রাক্ট ক্লাসকে ইন্টারফেস বলা যায় । অ্যাবস্ট্রাক্ট ক্লাসে অ্যাবস্ট্রাক্ট মেথড এবং রেগুলার মেথড দুটিই ছিল কিন্তু ইন্টারফেসে কোন প্রকার রেগুলার মেথড থাকবে না । ইন্টারফেসে কেবল অ্যাবস্ট্রাক্ট মেথডই থাকবে । চলুন আমরা একটি উদাহরণ দেখে নেইঃ

```

interface Animal{

    public abstract void name(String animalName);
    String color();
}

interface Cow{
    void work();
}

public class Main implements Animal, Cow{

    public static void main(String[] args) {

        Main m = new Main();
        m.name("I don't know this :P");
        System.out.println(m.color());
        m.work();
    }

    @Override
    public void name(String animalName) {
        System.out.println(animalName);
    }

    @Override
    public String color() {
        return "Red";
    }

    @Override
    public void work() {
        System.out.println("Gives Milk");
    }
}

```

লক্ষ করুন এখানে `interface` কিওয়ার্ডটি দিয়ে দুটি ইন্টারফেস ডিক্লেয়ার করা হয়েছে যথাক্রমে `Animal` এবং `Cow` । `Animal` ইন্টারফেসের মধ্য দুটি মেথড আছে যাদের একজনে `public` এবং `abstract` ডিক্লেয়ার করা হয়েছে কিন্তু অন্য মেথডটি কেবল রিটার্নটাইপ দেওয়া হয়েছে । এটির কারন হল ইন্টারফেসের মাঝে আপনি যদি কোন মেথডের পূর্বে `public` এবং `abstract` ডিক্লেয়ার নাও করেন তবু তারা বাই ডিফল্ট পাবলিক এবং অ্যাবস্ট্রাক্ট । এবার আসি `Main` ক্লাসে । এতক্ষন আমরা যেনে এসেছি যে জাভা মাল্টিপল ইনহেরিট্যান্স সাপোর্ট করেনা তাহলে এখানে কেন দুটি ইন্টারফেস ইমপ্লিমেন্ট করছে ? হ্যা সেটাই করবে কারন পরে ব্যাখ্যা করা হবে । এখানে লক্ষনীয় বিষয় হল ইন্টারফেসকে কিন্তু `implements` কিওয়ার্ড দিয়ে ইমপ্লিমেন্ট করতে হয় । এখানে কিন্তু এক্সটেন্ড হবেনা । একটি ক্লাস কেবল অন্য একটি ক্লাসকে এক্সটেন্ড করতে পারবে তবে একই সাথে অন্য গুন্য , এক বা একাধিক ইন্টারফেসকেও ইমপ্লিমেন্ট করতে পারবে । একাধিক ইন্টারফেস ইমপ্লিমেন্ট করার প্রয়োজন হলে কমা ( , ) দিয়ে একটির পর আরেকটি যোগ করতে হবে । তবে অবশ্যই অ্যাবস্ট্রাক্ট মেথড ইমপ্লিমেন্ট করতে ভুলবেন না । :P

কয়েকটি বিষয় জেনে রাখা ভালোঃ

১) একটি ক্লাস একটি মাত্র ক্লাস বা অব্যবস্থা ক্লাসকে এক্সটেন্ড করতে পারবে । ২) একটি ক্লাস বা অব্যবস্থা ক্লাস এক বা একাধিক ইন্টারফেসকে ইমপ্লিমেন্ট করতে পারবে । ৩) একটি ইন্টারফেস এক বা একাধিক ইন্টারফেসকে এক্সটেন্ড (ইমপ্লিমেন্ট নয় কিন্তু) করতে পারবে ।

উল্লেখ্য অব্যবস্থা মেথডের মত ইন্টারফেসেরও কোন অবজেক্ট ক্রিয়েট করা যায়না । যায়না সেটা বলা ভুল তবে সরাসরি যায়না । কিভাবে যায় সেটা বোঝার জন্য আপনাদের পলিমরফিজম পর্যন্ত যাওয়া লাগবে । ;)

### ## পলিমরফিজম (Polymorphism)

এবার আমরা কথা বলবো পলিমরফিজম নিয়ে। শব্দটির মধ্যেই একটি বিশেষ গাভীর্য় আছে যা কিনা একটি সাধারণ কথোপকথনকে অনেক গুরুত্বপূর্ণ করে তুলতে পারে। তবে এটি অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর একটি বহুল ব্যবহৃত কৌশল। এই শব্দটির সহজ মানে হচ্ছে যার একাধিক রূপ আছে অর্থাৎ বহুরূপিতা।

সহজ কথায় পলিমরফিজম হল এমন একটি টেকনিক বা পদ্ধতি যেখানে আমরা একটি ক্লাস, অ্যাবস্ট্রাক্ট ক্লাস বা ইন্টারফেসের অবজেক্ট ক্রিয়েট করি তার চাইল্ড ক্লাসের কনস্ট্রাক্টরের মাধ্যমে। অর্থাৎ আমরা একটি ক্লাসের অবজেক্ট ক্রিয়েট করবো অন্য একটি ক্লাসের কনস্ট্রাক্টর কল করে। সহজ ভাষায় এটিই হল পলিমরফিজম।

মনে করা যাক,

```
public class Liquid {
    public void swirl(boolean clockwise) {
        // Implement the default swirling behavior for liquids
        System.out.println("Swirling Liquid");
    }
}
```

এখন এর একটি অবজেক্ট তৈরি করতে চাইলে – আমাদের new অপারেটর ব্যবহার করে তা একটি ভেরিয়েবল এ রাখতে হবে।

```
Liquid myFavoriteBeverage = new Liquid ();
```

এখানে myFavoriteBeverage হচ্ছে আমাদের ভেরিয়েবল যা Liquid অবজেক্ট এর রেফারেন্স। আমরা এখন পর্যন্ত যা যা শিখেছি সে অনুযায়ী এই স্টেটমেন্টটি যথার্থ। তবে আমরা এর আগের অধ্যায়ে Is-A সম্পর্কে জেনে এসেছি।

আমাদের জাভা প্রোগ্রামিং পলিমরফিজম সাপোর্ট করায় আমরা myFavoriteBeverage এই রেফারেন্সের যায়গায় Is-A সম্পর্কিত যে কোন টাইপ রাখতে পারি। যেমন –

```
Liquid myFavoriteBeverage = new Coffee();
Liquid myFavoriteBeverage = new Milk();
```

এখানে Coffee এবং Milk হচ্ছে Liquid এর সাব-ক্লাস বা টাইপ এবং Liquid এদের সুপার ক্লাস বা টাইপ।

পলিমরফিজম নিয়ে আরও একটু আশ্চর্য হতে চাইলে আমরা এখন একটি বিষয় জানবো যা দিয়ে আমরা কোন একটি অবজেক্ট এর কোন মেথড কল করবো তবে তা কোন ক্লাসের অবজেক্ট সেটি না জেনেই। আরেকটু পরিষ্কার করে বলি, আমরা যখন সুপার ক্লাসের এর রেফারেন্স ধরে কোন এর মেথড কল করবো তখন কিন্তু আমরা জানি না যে এটি আসলে কোন অবজেক্ট এর মেথড। যেমন-

```
Liquid myFavoriteBeverage = // ...
```

এখানে আমাদের myFavoriteBeverage এই রেফারেন্স এ Liquid , Coffee , Milk এর যেকোন একটির অবজেক্ট হতে পারে। উদাহরণ -

```
public class Coffee extends Liquid {
    @Override
    public void swirl(boolean clockwise) {
        System.out.println("Swirling Coffee");
    }
}

public class Milk extends Liquid{
    @Override
    public void swirl(boolean clockwise) {
        System.out.println("Swirling Milk");
    }
}

public class CoffeeCup {
    private Liquid innerLiquid;

    void addLiquid(Liquid liq) {
        innerLiquid = liq;
        // Swirl counterclockwise
        innerLiquid.swirl(false);
    }
}
```

আমরা এখানে একটি CoffeeCup ক্লাস লিখেছি যার মাঝে addLiquid() নামে একটি মেথড আছে যা কিনা একটি Liquid টাইপ parameter নেয়, এবং সেই Liquid এর swirl() মেথড-কে কল করে।

কিন্তু আমরা আমাদের সত্যিকারের জগতে একটি কফি-কাপ এ শুধুমাত্র কফি-ই এড করতে পারি তা নয়, আমরা চাইলে যে কোন ধরনের লিকুইড এড করতে পারি, সেটি মিল্ক ও হতে পারে। তাহলে এই addLiquid মেথড তো শুধুমাত্র Liquid টাইপ parameter নেয়, তাহলে আমাদের সত্যিকারের জগতের সাথে এই প্রোগ্রামিং মডেল এর সাদৃশ্য থাকলো কোথায় ?

তবে মজার ব্যাপার এখানেই, আমাদের এই CoffeeCup ক্লাসটি পলিমরফিজমের ম্যাজিক ব্যবহার করে সত্যিকার অর্থেই আমাদের সত্যিকারের জগতের CoffeeCup এর মতোই কাজ করে।

```

public class MainApp {
    public static void main(String[] args) {
        // First you need a coffee cup
        CoffeeCup myCup = new CoffeeCup();

        // Next you need various kinds of liquid
        Liquid genericLiquid = new Liquid();
        Coffee coffee = new Coffee();
        Milk milk = new Milk();

        // Now you can add the different liquids to the cup
        myCup.addLiquid(genericLiquid);
        myCup.addLiquid(coffee);
        myCup.addLiquid(milk);
    }
}

```

উপরের কোড গুলোতে দেখা যাচ্ছে যে আমরা একটি `CoffeeCup` এর একটি অবজেক্ট তৈরি করে সেটিতে বিভিন্ন রকম `Liquid` এড করতে পারছি।

আরেকটু লক্ষ্য করি,

```

void addLiquid(Liquid liq) {
    innerLiquid = liq;
    // Swirl counterclockwise
    innerLiquid.swirl(false);
}

```

এই মেথডটিতে `innerLiquid.swirl(false)` যখন কল করি তখন কিন্তু আমরা জানি না যে এই `innerLiquid` আসলে কোন অবজেক্ট এর রেফারেন্স। এটি লিকুইড বা এর যে কোন সাব-টাইপ হতে পারে।

কিছু প্রয়োজনীয় তথ্য-

১. একটি সাব ক্লাস এর অবজেক্টকে আমরা এর সুপার ক্লাসের রেফারেন্স এ এসাইন করতে পারি। ২. সাব ক্লাসের অবজেক্টকে সুপার ক্লাসের রেফারেন্স-এ এসাইন করলে, মেথড কল করার সময় শুধু মাত্র সুপার ক্লাসের মেথড গুলোকেই কল করতে পারি। ৩. তবে সাব ক্লাস যদি সুপার ক্লাসের মেথড অডাররাইড করে, তাহলে যদিও আমরা সুপার ক্লাস এর রেফারেন্স ধরে মেথড কল করছি, কিন্তু রানটাইম-এ সাব ক্লাসের মেথডটি কল হবে। মনে রাখতে হবে এটি শুধুমাত্র মেথড অডাররাইড করা হলেই সত্য হবে।

**আপ-কাস্টিং(Upcasting ) এবং ডাউনকাস্টিং (Downcasting)**

```

Liquid liquid = new Coffee ();

```

এখানে সাব ক্লাসের অবজেক্টকে সুপার ক্লাসের রেফারেন্স এ এসাইন করা হয়েছে। একে বলা হয় আপ-কাস্টিং। এই কাস্টিং সবসময় সেইফ ধরা হয় কারণ আপকাস্টিং এর ক্ষেত্রে সাব ক্লাস সবসময়ই সুপার ক্লাসের সবকিছু ইনহেরিট করে এবং কম্পাইলার কম্পাইল করার সময়-ই এ কাস্টিং করা সম্ভব কিনা তা চেক করে থাকে।

```
Liquid liquid = new String();
```

উপরের স্টেটমেন্টটি কম্পাইলার কম্পাইল করবে না, কারণ String মোটেই Liquid ক্লাসের সাব ক্লাস নয়। এক্ষেত্রে কম্পাইলার incompatible types এর দেখাবে।

### হোমোজিনিয়াস কালেকশন ( Homogeneous Collection ):

হোমোজিনিয়াস কালেকশন হল একই ক্লাসের কিছু সংখ্যক অবজেক্টের কালেকশন। একটি উদাহরণ দিয়ে বিষয়টি একটু সুরাহা করা যাকঃ

```
interface Animal {

    public abstract void name(String animalName);
}

class Cow implements Animal {

    private String animalName;

    public void work(String animalWork) {
        System.out.println("Work of " + this.animalName + " is " + animalWork);
    }

    @Override
    public void name(String animalName) {
        this.animalName = animalName;
        System.out.println("Name of the animal is: " + this.animalName);
    }
}

public class Main {

    public static void main(String[] args) {

        Animal[] collection1 = new Cow[3];
        collection1[0] = new Cow();
        collection1[1] = new Cow();
        collection1[2] = new Cow();

        Cow[] collection2 = new Cow[3];
        collection2[0] = new Cow();
        collection2[1] = new Cow();
        collection2[2] = new Cow();
    }
}
```



লক্ষ করুন । এখানে Cow ক্লাসটি Animal ইন্টারফেসের চাইল্ড । এবং Main ক্লাসের main মেথড এর মাঝে ২ টি অবজেক্টের অ্যারে ডিক্লেয়ার করা হয়েছে । একটি Animal ক্লাসের অবজেক্টের অ্যারে যেটির সবগুলো অবজেক্ট Cow ক্লাসের কনস্ট্রাক্টর দিয়ে ইন্সট্যানশিয়েট করা হয়েছে । এখানে পলিমরফিজম স্পষ্ট । এবং অন্যটি অবজেক্ট অ্যারেটি চীরাচরিত অবজেক্ট অ্যারে । এই দুই অ্যারেই হল হোমোজিনিয়াস কালেকশনের উদাহরন । বোঝা যায়নি ? ওকে, এখানে collection1 অ্যারেটির প্রতিটি অবজেক্টই Cow ক্লাসের কনস্ট্রাক্টর দিয়ে ইন্সট্যানশিয়েট করা হয়েছে । তার মানে collection1 এর মাঝে সবগুলো অবজেক্টই একই ধরনের । যেহেতু এই অ্যারেটির সবগুলো এলিমেন্ট একই ধরনের/ক্লাসের অবজেক্ট সুতরাং এটিকে বলা হবে হোমোজিনিয়াস কালেকশন । একই কথা collection2 এর ক্ষেত্রেও প্রযোজ্য ।

### হেটারোজিনিয়াস কালেকশন ( Heterogeneous Collection ):

ভিন্নধর্মী অবজেক্টের কালেকশনকেই বলা হয় হেটারোজিনিয়াস কালেকশন । হেটারোজিনিয়াস কালেকশন বুঝতে হলে আমাদের একটি উদাহরন দেখে নেওয়া উত্তমঃ

```
class Animal {

    String animalName ;
    public Animal(String animalName){
        this.animalName = animalName;
    }
    public void name(){
        System.out.println("Animal name is: "+this.animalName);
    }
}

class Cow extends Animal {

    public Cow(String animalName) {
        super(animalName);
    }

    public void work(String animalWork) {
        System.out.println("Work of " + this.animalName + " is " + animalWork);
    }
}

class Dog extends Animal {

    public Dog(String animalName) {
        super(animalName);
    }

    public void work(String animalWork) {
        System.out.println("Work of " + this.animalName + " is " + animalWork);
    }
}

class Cat extends Animal {

    public Cat(String animalName) {
```

```

        super(animalName);
    }

    public void work(String animalWork) {
        System.out.println("Work of " + this.animalName + " is " + animalWork);
    }
}

public class Main {

    public static void main(String[] args) {

        Animal[] animals = new Animal[4];
        animals[0] = new Animal("Dolphin");
        animals[1] = new Cow("Big Cow");
        animals[2] = new Dog("Red Dog");
        animals[3] = new Cat("White Cat");
    }
}

```

খুব ভালোভাবে লক্ষ করুন । আমরা `Animal` ক্লাসের অবজেক্টের একটু অ্যারে ডিক্লেয়ার করেছি যার সাইজ ৪ । কিন্তু ইন্সট্যানশিয়েট করার সমস আমরা পলিমরফিজম মেকানিজম ব্যবহার করে এর চাইল্ড ক্লাসের ভিন্ন ভিন্ন কনস্ট্রাক্টর দিয়ে ইন্সট্যানশিয়েট করেছি । অর্থাৎ `animals` অ্যারেটির প্রতিটি অবজেক্টই আলাদা আলাদা কনস্ট্রাক্ট দিয়ে ইন্সট্যানশিয়েট করা এবং তাদের বিহ্যভিয়েরাল পার্থ্য আছে । এধরনের কালেকশনকে বলা হয় হেটারোজিনিয়াস কালেকশন ।

এবার একটু ভিন্ন পন্থায় এগোন যাক । মেইন ক্লাসটিকে আমরা একটু মডিফাই করবো । বাকী সবই ঠিক থাকবে আগের মত ।

```

public class Main {

    public static void main(String[] args) {

        Animal animal = new Cat("Cute Cat");
        animal.name();
        //animal.work("Some Work");//Not possible
        Cat cat = new Cat("Preety Cat");
        cat.name();
        cat.work("It plays");
    }
}

```

খেয়াল করে দেখুন আমরা `Animal` এবং `Cat` এর অবজেক্ট ক্রিয়েট করার সময় কনস্ট্রাক্টর ব্যবহার করেছি `Cat` এর কিন্তু `Animal` এর অবজেক্ট থেকে আমরা `work` মেথডটি কোন ভাবেই কল করতে পারছি না বা পারবো না কিন্তু `Cat` এর অবজেক্ট থেকে ঠিকই পারছি । কারনটা কি ? কারন হল `Animal` ক্লাসের মাঝে ঠিক যে যে মেথড

আছে সেগুলোকেই আমরা অ্যাক্সেস করতে পারব তবে `Cat` এর ইমপ্লিমেন্টেশন দিয়ে । `Animal` এর মাঝে নেই কিন্তু `Cat` ক্লাসে বাড়তি আছে এমন কোন মেথডকে আমরা অ্যাক্সেস করতে পারবো না । এমনকি `Animal` ক্লাসের অবজেক্টে `Cat` ক্লাসের `work` মেথডের কোন রেফারেন্সই ক্রিয়েট হবেনা ।

তাহলে এটা করি কেন আমরা ? এটা করার পেছনে বেশ কিছু কারন থাকতে পারে । প্রথমত আমরা প্যারেন্ট ক্লাস এবং চাইল্ড ক্লাসের ইমপ্লিমেন্টেশন নিয়ে কাজ করতে চাইলে পলিমরফিজমের এই সুবিধাটি নেওয়া হয় । অন্য কারনটি হল মেমোরি কনজাম্পশন । ভেবে দেখুন যদি `Animal` ক্লাসে ৩ টি মেথড থাকে যেগুলার জন্য আপনি `Cat` ক্লাসের ইমপ্লিমেন্টেশন ব্যবহার করতে চান , কিন্তু `Cat` ক্লাসের মাঝে ১৫ টির মত মেথড আছে এবং অনেক অ্যাট্রিবিউট । আপনি যদি `Cat` এর অবজেক্ট ক্রিয়েট করেন তবে মেমোরি থেকে প্রচুর স্পেস কনজিউম করবে উক্ত অবজেক্ট । অন্যদিকে আপনি যদি `Animal` এর অবজেক্ট ক্রিয়েট করেন `Cat` এর কনস্ট্রাক্টর ব্যবহার করে তাহলে `Cat` ক্লাসের ইমপ্লিমেন্টেশন ব্যবহার করতে পারছেন এবং মেমোরি থেকে খুব কম মেমোরি কনজিউম করছে ( `Animal` মেথডগুলার জন্য প্রয়োজনীয় মেমোরি মাত্র ) । কোনটি বেশি সুবিধাজনক ? এছাড়া আরো কারন আছে । পরবর্তীতে সেগুলো নিয়েও আলোচনা করা হবে ।

চলবে .....

## ইনক্যাপসুলেশান (Encapsulation)

আমরা এতোক্ষণে জেনে ফেলেছি যে, একটি অবজেক্ট হলো কতগুলো ডাটা এবং মেথড এর সমষ্টি। অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর আরেকটি খুবই গুরুত্বপূর্ণ বিষয় আছে, যা হচ্ছে, একটি ক্লাসের মধ্যে ডাটা গুলোকে লুকিয়ে রাখা এবং শুধুমাত্র মেথডের মাধ্যমে সেগুলোকে একসেস করতে দেওয়া। এর নাম হচ্ছে ইনক্যাপসুলেশান(Encapsulation)। এর মাধ্যমে আমরা সব ডাটা গুলোকে ক্লাসের মধ্যে সিল করে একটা কেপসুলের মধ্যে রেখে দিই এবং সেগুলো শুধুমাত্র যেসব মেথড গুলোকে ট্রাস্ট করা যায়, তাদের মাধ্যমে একসেস করতে দিই।

তবে এই এতো প্রোটেকশান এর কারণ কি হতে পারে তা যদি একটু জেনে নিই শুরুতে তাহলে আমার মনে খুব ভাল হয় –

যারা অনেক লেখালেখি করে এমনকি যারা কোড লিখে তারাও জানে যে, একটা লেখা ততই ভাল হয় সেটাকে যত বেশি রি-রাইট করা হয়। আপনি যদি একটা কোড লিখে ফেলে রাখেন এবং কিছুদিন পরে আবার সেটি খুলে দেখেন- দেখা যাবে যে আপনি আরও একটি ভাল উপায় বের পেয়ে যাবেন সেই কোডটি লেখার। এটি সব সময়ই হয়। এই বার বার কোড চেঞ্জ করে নতুন করে লেখাকে বলা হয় রিফেক্টরিং(refactoring)। আমরা একটি কোডকে বার বার লিখে সেটাকে আরও বেশি কিভাবে সহজবোধ্য কোড লেখার চেষ্টা করি যাতে সেই কোডটি আরও ভালভাবে মেইনটেইন করা যায়।

কিন্তু এখানে একটি চিন্তার বিষয় হচ্ছে। আমরা জানি যে অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর মাধ্যমে আমরা যে সফটওয়্যার সিস্টেম তৈরি করি তাতে নানা রকম অসংখ্য অবজেক্ট থাকে যেখানে একটি অবজেক্ট আরেকটির সাথে তথ্য আদান প্রদান করে, একটি আরেকটির উপর নির্ভর করে কাজ করে থাকে। ধরা যাক, A একটি অবজেক্ট যার উপর B নির্ভর করে। ধরা যাক B এখানে কনজুমার অবজেক্ট। এখন আমাদের যদি A কে কোন রকম পরিবর্তন করতে চাই, তাহলে B আগের মতোই থাকতে চাইবে। এখানে দুটো অবজেক্ট হয়তো দুইজন ভিন্ন প্রোগ্রামার লিখে থাকতে পারে। সুতরাং একে অন্যের পরিবর্তন নিয়ে যাতে সমস্যা পরতে না হয়, সেই ব্যবস্থা করতে হবে। আমরা অনেক সময় নানা রকম লাইব্রেরি ব্যবহার করতে হয় বিভিন্ন প্রজেক্টে এবং এগুলোর উপর নির্ভর করে করে আমাদের প্রজেক্ট দাড়িয়ে যায়। এই লাইব্রেরি গুলোর মাঝেই মাঝেই ডার্সন পরিবর্তন হয়। কিন্তু মজার ব্যাপার হলো এগুলো পরিবর্তন হলেও আমাদের কোড নতুন করে লিখতে হয় না। আবার অন্যদিকে লাইব্রেরি যারা তৈরি করে তাদেরও এই কোড পরিবর্তনের স্বাধীনতা থাকা চাই, কিন্তু সক্ষেত্রে যাতে আমাদের প্রজেক্ট এর কোন সমস্যা যাতে না হয় সেটাও মনে রাখতে হবে।

তো এই সমস্যা সমাধানের একটি উপায় আছে। সেটি হলো- লাইব্রেরি কোড-এর যে মেথড গুলো আছে সেগুলো মোটেও রিমুভ করা যাবে না। কারণ আমরা যখন একটি লাইব্রেরির একটি নির্দিষ্ট ক্লাসের মেথড নিয়ে কাজ করবো, আমরা চাইবো না কোন ভাবেই আমাদের কোড ভেঙ্গে যাক। লাইব্রেরির প্রোগ্রামার সেই ক্লাস নিয়ে যা কিছু করতে পারবে, কিন্তু আমরা যে সব মেথড ব্যবহার করেছি সেগুলোকে মুছে ফেলতে পারবে না। তারপর ফিল্ড বা প্রোপার্টিজ এর ক্ষেত্রেও লাইব্রেরি যে লিখেছে সে কিভাবে জানবে যে কোন ফিল্ড বা প্রোপার্টিজ গুলো আমরা আমাদের প্রজেক্ট এর ক্ষেত্রে একসেস করেছি? কোন ভাবেই জানার উপায় নেই। কারণ আমরা আমাদের কোড কিভাবে করেছি যা লাইব্রেরি যে লিখেছে তার জানার কথা নয়। কিন্তু যে প্রোগ্রামার লাইব্রেরি লিখেছে সে সবসময়ই চাইবে তার কোড এ নতুন কিছু এড করতে, আগের থেকে ভাল করা ইত্যাদি। এই সমস্যা সমাধানের জন্যে জাভা আমাদেরকে কতগুলো একেসেস স্পেসিফায়ার (access specifiers) দিয়ে থাকে, যার মাধ্যমে লাইব্রেরি প্রোগ্রামার ঠিক করতে পারে যে কোড এর

কোন কোন অংশ গুলো আমরা যখন আমাদের প্রজেক্ট এ ব্যবহার করতে পারবো আর কোন কোন গুলো করতে পারবো না। এতে সুবিধা হচ্ছে, লাইব্রেরি প্রোগ্রামার সে সব অংশ গুলো আমাদেরকে ব্যবহার করতে দিচ্ছে, সেই অংশ গুলোতে ইচ্ছে মতো পরিবর্তন/পরিবর্তন করতে পারবে কোন রকম চিন্তাভাবনা ছাড়া।

আমরা যখন একটা বড় সিস্টেমে কাজ করি আমাদের নানা রকম অবজেক্ট লিখতে হয়। একটি অবজেক্ট আরেকটি অবজেক্ট কে ব্যবহার করে। এই একসেস প্রটেকশানের মাধ্যমে আমরা নির্ধারণ করে দিতে পারি যে একটি নির্দিষ্ট অবজেক্ট এর কোন অংশ গুলো অন্য অবজেক্ট ব্যভহার করতে পারবে, আর কোন গুলো পারবে না। এতে উপরের সমস্যার সমাধান হয়ে যায়। এছাড়াও আরেকটি ব্যাপার হয়। আমরা যখন কোন একটি ক্লাস নিয়ে কাজ করতে যাবো, সেই অবজেক্ট-এ হাজার লাইন কোড থাকে পারে। পুরটা একেবারে দেখতে গেলে আমরা হয়তো কনফিউজড হয়ে যাবো কিংবা খুব কমপ্লেক্স কোড হলে বুঝতে অসুবিধা হতে পারে। কিন্তু সেই কোড যদি এমন ভাবে করা থাকে যেখানে অল্প অংশ আমাদের ব্যবহারের জন্যে অপেন করা থাকে, বাকি গুলো হাইড করা থাকে তাহলে আমরা যে অংশটুকু হাইড করা সেই অংশ নিয়ে চিন্তা করতে হবে না। এই কোড হাইড করার ঘটনাকে অবজেক্ট ওরিয়েন্টেড প্রোগ্রামিং এর ভাষায় এনক্যাপসুলেশন(Encapsulation) বলা হয়।

জাভাতে তিনটি একসেস কন্ট্রোল করার জন্যে তিনটি কি ওয়ার্ড আছে। সেগুলো হল- `Public`, `protected` এবং `private` এখন আমরা বিভিন্ন রকম একসেস কন্ট্রোল দেখাবো-

## Default Access

এর মানে হচ্ছে আমরা যদি কোন কি-ওয়ার্ড ব্যবহার না করি তাহলে সেটি Default Access আর মাঝে পরে। কোন ক্লাস এর ডেরিয়েবল বা মেথড এর আগে যদি কোন একসেস মডিফায়ার না থাকে তাহলে সেই ক্লাসটি যে প্যাকেজের মধ্যে আছে সেই প্যাকেজ এর সব ক্লাস থেকে একসেস করা যাবে।

```
package bd.com.howtocode.java;

import java.util.Random;

public class HelloWorld {
    String version = "2.56";

    int getRandomInt() {
        return new Random().nextInt();
    }
}
```

এই ক্লাসের ডেরিয়েবল version এবং getRandomInt() মেথড কে bd.com.howtocode.java এই প্যাকেজ এর সকল ক্লাস একসেস করতে পারবে।

## Private Access Modifier - `private` :

কোন ক্লাসের মেথড, ডেরিয়েবল, কনস্ট্যান্ট এর আগে যদি private কিওয়ার্ড থাকে তাহলে সেগুলোকে সেই ক্লাস ছাড়া অন্য কোন ক্লাস একসেস করতে পারবে না।

উদাহরণ-

```

package bd.com.howtocode.java;

public class User {
    private String name;
    private String emailAddress;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmailAddress() {
        return emailAddress;
    }

    public void setEmailAddress(String emailAddress) {
        this.emailAddress = emailAddress;
    }
}

```

এই ক্লাসের এর ভেরিয়েবল `name` এবং `emailAddress` কে কোন ভাবেই অন্য কোন ক্লাস থেকে একসেস করা যাবে না। কিন্তু আমরা যদি এদের কে একসেস করতে চাই তাহলে একসেসর মেথড ব্যবহার করতে পারি।

## Public Access Modifier - `public` :

কোন ক্লাসের মেথড, ভেরিয়েবল, কনস্ট্রাকটর এর আগে যদি `public` কিওয়ার্ড থাকে তাহলে সেগুলোকে যে কোন ক্লাস থেকে একসেস করা যায়।

```

public class Milk{
    public void swirl(boolean clockwise) {
        System.out.println("Swirling Milk");
    }
}

```

## Protected Access Modifier - `protected` :

কোন ক্লাসের মেথড, ভেরিয়েবল, কনস্ট্রাকটর এর আগে যদি `protected` কিওয়ার্ড থাকে তাহলে সেগুলোকে অন্য প্যাকেজ থেকে সেই ক্লাসের সাব ক্লাস একসেস করতে পারবে আর নিজের প্যাকেজ এর সবাই একসেস করতে পারবে।

```
class AudioPlayer {
    protected boolean openSpeaker(Speaker sp) {
        // implementation details
    }
}
```

একসেস লেভেল একটি টেবলি -

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

## পাঠ ৬: জাভা এক্সেপশান হ্যান্ডেলিং

- এক্সেপশান বেসিকস
- টাই ক্যাচ- ফাইনালি
- চেকড-এক্সেপশান
- আনচেকড-এক্সেপশান
- থ্রয়িং এক্সেপশান
- NullPointerException
- ArrayIndexOutOfBoundsException
- কিভাবে নিজস্ব এক্সেপশান লিখবো
- সারসংক্ষেপ

আমরা একটি প্রোগ্রাম লিখি যার একটি নরমাল ফ্লো থাকে, তবে কোন কারণে যদি এই ফ্লো ব্যাহত হয় তাহলে জাভা রানটাইম একটি ইভেন্ট ফায়ার করে, একে এক্সেপশান বলা হয়।

সহজ কথায় এক্সেপশন হচ্ছে এক ধরনের ইরর যা কিনা প্রোগ্রাম চলাকালীন সময়ে দেখা দিতে পারে।

একটি উদাহরণ দেখা যাক-

```
public class Main {  
  
    public static void main(String[] args) {  
        int a = 1;  
        int b = 0;  
  
        int result = divide(a, b);    // 1  
        System.out.println("Result: " + result);    // 2  
    }  
  
    public static int divide(int a, int b) {  
        return a / b;  
    }  
}
```

১. এখানে `divide()` মেথডটিতে `a` এবং `b` আর্গুমেন্ট পাস করা হলে মেথডটি প্রথম আর্গুমেন্টকে দ্বিতীয় আর্গুমেন্ট দিয়ে ভাগ করে ফলাফল `result` ভ্যারিয়েবল-টিতে এসাইন করবে।

২. এখানে `result` এর মান প্রিন্ট করা হবে।

আমরা যদি এই প্রোগ্রামটি রান করি তাহলে console এ নিচের আউটপুট-টি পাবো-



```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at com.bazlur.exception.Main.divide(Main.java:18)
    at com.bazlur.exception.Main.main(Main.java:13)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
    at java.lang.reflect.Method.invoke(Method.java:483)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:134)
```

এই আউটপুট থেকে আমরা বুঝতে পারি যে, আমাদের প্রোগ্রামটি-তে একটি সমস্যা হয়েছে এবং প্রোগ্রামটি এখানেই থেমে গেছে, `System.out.println("Result: " + result);` এই লাইনটি এক্সিকিউট হয় নি।

এবার আমরা নিচের প্রোগ্রামটি রান করি-

```
public class Main {

    public static void main(String[] args) {
        int a = 1;
        int b = 0;

        int result = 0;
        try {
            result = divide(a, b);
        } catch (ArithmeticException e) {
            System.out.println("You can't divide " + a + " by " + b);
        }

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

    public static int divide(int a, int b) {
        return a / b;
    }
}
```

এবার **console** এ নিচের আউটপুটটি দেখবো -

```
You can't divide 1 by 0
```

```
Result: 0
```

এবার লক্ষ্য করুন। প্রোগ্রামটি কিন্তু থেমে যাই নি, বরং শূন্য দিয়ে যে কোন সংখ্যাকে ভাগ করা যাবে না, তার জন্য একটি মেসেজ প্রিন্ট করেছে এবং শেষ পর্যন্ত প্রত্যেকটি লাইন এক্সিকিউট হয়েছে।

এই প্রোগ্রামটিতে আমরা নতুন কিওয়ার্ড ব্যবহার করেছি, সেগুলো হলো- try, catch এবং এগুলো দিয়ে আমাদের যে কোড ব্লকটিতে ইরর হওয়ার সম্ভাবনা ছিল, সেই অংশটুকুকে wrap করেছি। এতে করে এই কোড ব্লক-এ যদি কোন ধরনের ইরর হয় তাহলে প্রোগ্রামটি catch ব্লক-এ চলে যায়, এবং এই ব্লক এর ইন্সট্রাকশন গুলো এক্সিকিউট করে এরপর নিচের কোড ব্লক এ চলে যায়।

আর এই প্রক্রিয়াকে আমরা এক্সেপশন হ্যান্ডেলিং বলি, অর্থাৎ প্রোগ্রাম এর কোন অংশে যদি কোন ধরনের এক্সেপশন বা ইরর হয় তাহলে আমাদের প্রোগ্রামটি যাতে বন্ধ না হয় যায় বরং সেইসব অবস্থায় ইউজারকে যাতে করে অর্থপূর্ণ মেসেজ দেওয়াতে এক্সেপশন হ্যান্ডেলিং বলে।

## The try Block

যদি কোন কোড ব্লক -এ যদি ইরর হওয়ার সম্ভাবনা থাকে তাহলে আমরা সেই কোড ব্লক-কে try ব্লক দিয়ে ইনক্লোজ করতে হয়। উদাহরণ-

```
try {
    code
}
catch and finally blocks . . .
```

এই try ব্লক এর মাঝে এক বা একাধিক লাইন কোড থাকতে পারে। catch এবং finally ব্লক পরের সেকশনে দেখানো হবে।

একটি উদাহরণ দেখা যাক-

```
private List<Integer> list;
private static final int SIZE = 10;

public void writeList() {
    PrintWriter out = null;
    try {
        System.out.println("Entered try statement");
        out = new PrintWriter(new FileWriter("file.txt"));
        for (int i = 0; i < SIZE; i++) {
            out.write(i);
        }

    } catch (IOException e) {
    }
}
```

উপরের প্রোগ্রামটিতে একটি মেথড আছে - writeList() যা কিনা একটি ফাইল এ একটি লিস্ট থেকে ড্যালা পড়ে তা রাইট করে। এই মেথড-টি তে একাধিক এক্সেপশন বা ইরর হতে পারে। যেমন -

out = new PrintWriter(new FileWriter("file.txt")); এই লাইনটিতে আমরা একটি ফাইল অপেন করার চেষ্টা করেছি। কিন্তু এই ফাইলটি সিস্টেমে নাও থাকতে পারে, কিংবা থাকলেও সেটি অপেন করা যাচ্ছে না ইত্যাদি। সেক্ষেত্রে আমাদের সিস্টেম IOException ত্রুটি করবে এবং প্রোগ্রামটি বন্ধ হয়ে যাবে। এছাড়াও আমরা একটি ফর

লুপ ব্যবহার করেছি, এক্ষেত্রে ফাইল এ রাইট করার সময়ও ইরর বা এক্সেপশন হতে পারে। তাই এইসব ইরর বা এক্সেপশন কে হ্যান্ডেল করার জন্যে আমরা কোড ব্লকটিকে `try` ব্লক এর ভেতরে রেখেছি। এখন প্রোগ্রামটি চলার সময় যদি কোন ইরর বা এক্সেপশন হয় তাহলে প্রোগ্রাম এক্সিকিউশন সেখান থেকেই `catch` ব্লক এ চলে যাবে।

## The catch Blocks

`try` ব্লক এর সাথেই `catch` ব্লক লিখতে হয়। তবে আমরা একটি `try` ব্লকের সাথে একাধিক `catch` ব্লক লিখতে পারি।  
উদাহরণ-

```
try {

} catch (ExceptionType name) {
// catch blog # 1
} catch (ExceptionType name) {
// catch blog # 1
}
```

`catch` কিওয়ার্ড এর সাথে প্যারেন্টসিস এর মাঝে আমরা আর্গুমেন্ট দিতে হয় যা কি টাইপ এক্সেপশন হ্যান্ডেল করা হচ্ছে তা নির্দেশ করে। এখানে `ExceptionType` একটি প্লেস হোল্ডার। এখানে যে কোন ক্লাস যা কিনা `Throwable` ক্লাস কে ইনহেরিট করে তা বসতে পারে।

`try` ব্লক এর কোন কোড-এ যদি কোন এরর বা এক্সেপশন হয় তাহলে প্রোগ্রামের এক্সিকিউশন পয়েন্ট `catch` ব্লকে চলে আসে এবং শুধুমাত্র তখনই `catch` ব্লক এর কোড এক্সিকিউট হয়।

যদি একাধিক `catch` ব্লক থাকে তাহলে এক্সেপশন এর টাইপ অনুযায়ী `catch` ব্লক সিলেকটেড হয়।

```
try {

} catch (IndexOutOfBoundsException e) {
    System.err.println("IndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

এখানে `try` ব্লকে যদি `IndexOutOfBoundsException` হয় তাহলে প্রথম `catch` ব্লকটি এক্সিকিউট হবে। আর যদি `IOException` হয় তাহলে পরের `catch` ব্লকটি এক্সিকিউট হবে।

জাভা ৭ এবং পরবর্তি ভার্সন গুলোর জন্যে একটি নতুন ফিচার আছে যাতে করে একটি `catch` ব্লক দিয়ে অনেকগুলো এক্সেপশন হ্যান্ডেল করা যায়। উদাহরণ -

```
catch (IOException|SQLException ex) {
    logger.log(ex);
}
```

এখানে catch ব্লক-এ একাধিক এক্সেপশন একটি ডার্টিকেল বার (|) দিয়ে আলাদা করা হয়।

## The finally Block

উপরের উদাহরণ গুলো থেকে দেখলাম যে, try ব্লক এর কোড -এ এক্সেপশন হলে শুধুমাত্র catch ব্লকের কোড গুলো এক্সিকিউট হয়। তবে আমাদের এমন কোন সিচুয়েশন থাকতে পারে যখন আমরা চাই ইরর হোক বা না হোক, একটি কোড ব্লক আমরা সবসময়ই এক্সিকিউট করতে চাই, তাহলে আমরা finally ব্যবহার করি।

```
public void openFile() {
    FileReader reader = null;
    try {
        reader = new FileReader("someFile");
        int i = 0;
        while (i != -1) {
            i = reader.read();
            System.out.println((char) i);
        }
    } catch (IOException e) {
        //do something clever with the exception
    } finally {
        if (reader != null) {
            try {
                reader.close();
            } catch (IOException e) {
                //do something clever with the exception
            }
        }
        System.out.println("--- File End ---");
    }
}
```

উপরের প্রোগ্রামটিতে আমরা একটি ফাইল অপেন করছি এবং কিছু কাজ করেছি। এজন্যে একটি FileReader ক্লাসের অবজেক্ট তৈরি করেছি। আমরা চাই এই FileReader অবজেক্টটি কাজ শেষ হয়ে গেলে ক্লোজ করতে। এক্ষেত্রে আমরা finally ব্লক এ আমাদের একই ক্লোজিং এর কোডটি লিখেছি। এতে করে এই সুবিধা হচ্ছে যে, আমাদের এই ট্রাই ব্লক-এর কোড কাজ করুক আর না করুক, শেষে আমাদের FileReader এর অবজেক্টটি ক্লোজ হয়ে যাবে।

অর্থাৎ আমরা শুধুমাত্র তখনই ফাইনালী ব্লক ব্যবহার করি যখন আমরা নো ম্যাটার হ্যাট, একটি কোড ব্লক সবসময়ই এক্সিকিউট করতে চাই।

## Identifying Exception Point

try, catch এবং finally ব্লক ব্যবহার করে এক্সেপশন হ্যান্ডেল করার সময় আমাদের যে বিষয়টির উপর বিশেষ গুরুত্ব দিতে হবে সেটি হল নির্দিষ্ট পয়েন্টেই কেবল try এবং catch ব্যবহার করা। ঠিক যেখানে এক্সেপশন ঘটে বা ঘটার সম্ভাবনা থাকবে সেখানেই কেবল আমাদের প্রপার এক্সেপশন হ্যান্ডেলিং থাকা জরুরী। অন্যথায় কোড রান করবে কিন্তু কাঙ্ক্ষিত ফলাফল পাওয়া যাবে না।

উদাহরন হিসাবে আমরা মনে করি আমাদের একটি মেথড লিখতে বলা হল যেটিতে একটি স্ট্রিং অ্যারে পাস করা হবে এবং সেই অ্যারে এর মাঝ থেকে যে স্ট্রিং গুলো ইন্টিজার নাম্বার রিপ্রেজেন্ট করে সেগুলার যোগফল রিটার্ন করতে হবে । আমরা যদি কোডটি এভাবে লিখিঃ

```
public class Main {

    public static void main(String[] args) {

        String[] strings = {"1", "2", "3", "4", "5", "6"};
        System.out.println(new Main().getSum(strings));
    }

    public int getSum(String[] strings){

        int result = 0;

        try {

            for (String string : strings) {
                result += (Integer.valueOf(string));
            }
        } catch (NumberFormatException e) {

            System.err.println(e);
        }

        return result;
    }
}
```

উপরের কোডটি 21 সংখ্যাটি প্রিন্ট করবে যেটি getSum নামক মেথডটি রিটার্ন করছে । কিন্তু আমরা যদি ইনপুট স্ট্রিংটি একটু মডিফাই করে String[] strings = {"1", "2", "3", "four", "5", "6"}; করে দেই তাহলে প্রথমে প্রিন্ট করবে 6 এবং তারপর প্রিন্ট করবে java.lang.NumberFormatException: For input string: "four" । কিন্তু প্রবলেম অনুযায়ী প্রিন্ট করা কথা ছিল 17 । কারন four বাদ দিলে বাকী যতগুলো স্ট্রিং ইন্টিজার নাম্বার রিপ্রেজেন্ট করে সেগুলার যোগফল । সেক্ষেত্রে আমরা যদি কোডটি একটু মডিফাই করে ঠিক যেখানে এক্সেপশন হওয়া সম্ভব সেখানেই try ব্লকটি ব্যবহার করতাম তাহলে এই সমস্যা থেকে মুক্তি পাওয়া সম্ভব ছিল । কোডটি যদি এভাবে করিঃ

```

public class Main {

    public static void main(String[] args) {

        String[] strings = {"1", "2", "3", "four", "5", "6"};
        System.out.println(new Main().getSum(strings));
    }

    public int getSum(String[] strings) {

        int result = 0;

        for (String string : strings) {

            try {

                result += (Integer.valueOf(string));
            } catch (NumberFormatException e) {

                System.err.println(e);
            }
        }

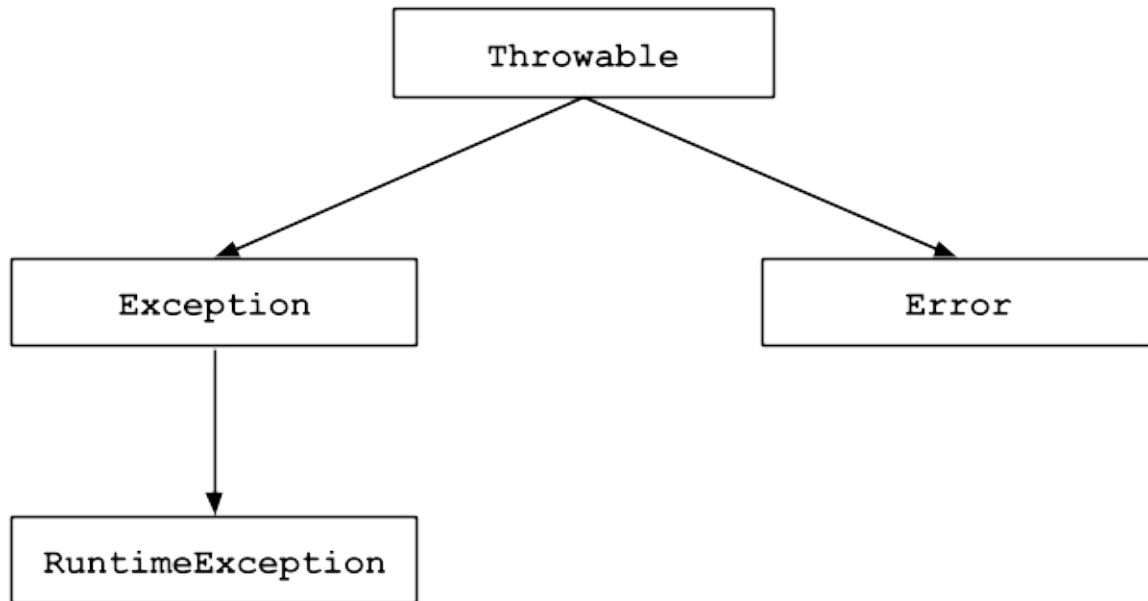
        return result;
    }
}

```

এবার যদি আমরা `String[] strings = {"1", "2", "3", "four", "5", "6"};` এই স্ট্রিংটি ইনপুট আকারে দেই তাহলে দেখবো একটা এক্সেপশন ঠিকই থ্রো করছে তবে রেজাল্ট হিসাবে আমরা যেটি চেয়েছিলাম সেটিও প্রিন্ট করছে । এভাবে আমরা ঠিক নির্দিষ্ট পয়েন্টে এক্সেপশন ডিটেক্ট করে হ্যান্ডেল করতে পারি । এতে করে ওভারঅল কোডের পারফরমেন্স যেমন বাড়বে তেমন কোড অনেক বেশি বাগফ্রী ও হবে ।

## Checked or Unchecked Exceptions

জাভাতে সব এক্সেপশন গুলো `Throwable` ক্লাসকে ইনহেরিট করে তৈরি । অর্থাৎ এক্সেপশন হাইআরকি এর একদপ উপরে এই `Throwable` ক্লাস এর অবস্থান । এর ঠিক নিচেই দুটি সাব ক্লাস হলে - `Exception` এবং অন্যটি হলো `RuntimeException` । এবং এই দুটি ক্লাস দুটি আলাদা শ্রেণীবিভাগের সূচনা করেছে । তবে এই শ্রেণীবিভাগের আরেকটি শাখা আছে, সেটি হলো - `Error` তবে এগুলো প্রোগ্রাম চলাকালিন সময়ে সাধারণত ধরা হয় না । এগুলো মূলত জাভা রানটাইম সিস্টেম নিজে থেকে হ্যান্ডেল করে এবং এটি আমাদের এই বইয়ের আলোচনার বাইরে ।



```
public class ExceptionDemo5 {  
  
    public void fetchData(String url) {  
        try {  
            String data = fetchDataFromUrl(url);  
        } catch (CheckedException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public String fetchDataFromUrl(String url) throws CheckedException {  
        if (url == null) {  
            throw new CheckedException("Url Not found");  
        }  
  
        String data = null;  
        //read lots of data over HTTP and return  
        //it as a String instance.  
  
        return data;  
    }  
}
```

```
public class ExceptionDemo6 {  
    public void fetchData(String url) {  
        String data = fetchDataFromUrl(url);  
    }  
  
    public String fetchDataFromUrl(String url) {  
        if (url == null) {  
            throw new UncheckedException("Url Not found");  
        }  
  
        String data = null;  
        //read lots of data over HTTP and return  
        //it as a String instance.  
  
        return data;  
    }  
}
```

```
public class CheckedException extends Exception {  
    public CheckedException(String message) {  
        super(message);  
    }  
}
```

```
public class UncheckedException extends RuntimeException {  
    public UncheckedException(String message) {  
        super(message);  
    }  
}
```



## এক্সেপশান হ্যান্ডেলিং: চলুন আরও একটু গভীরভাবে পর্যবেক্ষণ করি

আমরা ইতিমধ্যে জেনে ফেলেছি যে, একটা সিস্টেম এ নানা রকম সমস্যা হতে পারে। একটি প্রোগ্রাম চলতে গিয়ে হঠাৎ করে থেমে যেতে পারে কিংবা ক্র্যাশ করতে পারে। কিন্তু আমরা যখন একটি প্রোগ্রাম লিখি, আমরা অবশ্যই চাই প্রোগ্রামটি ভাল ভাবে চলুক কোন রকম সমস্যা ছাড়াই। কিন্তু সমস্যা হতেই পারে এবং এর জন্যে আমাদের প্রস্তুত হয়ে থাকাকাটা জরুরী।

শুরুতে একটি টার্ম সম্পর্কে পরিচয় করিয়ে দিই - Fault-tolerant

ফল্ট টলারেট যার বাংলা হতে পারে সমস্যা সহিষ্ণু। আমরা যেহেতু জানি যে আমাদের প্রোগ্রাম-এ সমস্যা হতে পারে, এবং আমরা চাই যে সখন সমস্যাটি হবে- তখনও প্রোগ্রামটি বন্ধ না হয়ে অন্য কোন ভাবে চলতে থাকে। আমরা যদি এমন ভাবে প্রোগ্রামটি লিখতে পারি তাহলে সেই প্রোগ্রামকে ফল্ট টলারেট প্রোগ্রাম লিখবো।

মনে করা যাক – আমাদের দেশে একটা সময় প্রতি ঘণ্টায় একবার করে চলে পাওয়ার যেত। এখন যদি কোন সিস্টেম তৈরি করি যা পাওয়ার এর উপর নির্ভরশীল, তাহলে যখন পাওয়ার থাকবে না, তখন সিস্টেমটি কাজ করবে না। এজন্যে আমরা বিকল্প ব্যবস্থা হিসেবে জেনারেটর রাখতে পারি, যাতে করে যখন মেইন পাওয়ার লাইন থাকবে না, তখন জেনারেটরের মাধ্যমে আমাদের সিস্টেমটি চলতে থাকবে। এই সিস্টেমটিকে আমরা তখন ফল্ট টলারেট সিস্টেম বলবো।

তো আমাদের এই টপিক এর উদ্দেশ্য হচ্ছে আমরা কিভাবে ফল্ট টলারেট জাভা প্রোগ্রাম লিখতে পারি।

শুরুতে আমরা একটি প্রোগ্রাম দেখি যাতে এক্সেপশান হ্যান্ডেলিং ব্যবহার করা হয় নি।

নিচের প্রোগ্রামটি রান করুন-

```
public class DivideByZeroNoExceptionHandling {
    public static int divide(int a, int b) {
        return a / b;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Please enter an integer: ");
        int a = scanner.nextInt();
        System.out.println("Please enter another integer: ");
        int b = scanner.nextInt();

        int result = divide(a, b);

        System.out.println(String.format("Result: %d/%d = %d", a, b, result));
    }
}
```

**Take 1**

```
Please enter an integer:
100
Please enter another integer:
45
Result: 100/45 = 2
```

**Take# 2**

```
Please enter an integer:
100
Please enter another integer:
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at com.bazlur.tips.DivideByZeroNoExceptionHandling.divide(DivideByZeroNoExceptionHand
    at com.bazlur.tips.DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandli
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
    at java.lang.reflect.Method.invoke(Method.java:498)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)
```

**Take # 3**

```
Please enter an integer:
100
Please enter another integer:
bazlur
Exception in thread "main" java.util.InputMismatchException
    at java.util.Scanner.throwFor(Scanner.java:864)
    at java.util.Scanner.next(Scanner.java:1485)
    at java.util.Scanner.nextInt(Scanner.java:2117)
    at java.util.Scanner.nextInt(Scanner.java:2076)
    at com.bazlur.tips.DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandli
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
    at java.lang.reflect.Method.invoke(Method.java:498)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)
```

- Take #1 এ প্রোগ্রামটি খুব ভাল ভাবে রান করছে।
- Take #2 তে ইনপুট হিসেবে শূন্য দেওয়াতে আমাদের প্রোগ্রামটি ঠিকভাবে কাজ করেনি বরং অনেকগুলো লাইন প্রিন্ট করেছে।

- Take #3 তে ইনপুট হিসেবে ইন্টিজার এর পরিবর্তে স্ট্রিং দেওয়ায় প্রোগ্রামটি কাজ করে নি, বরং অনেকগুলো লাইন প্রিন্ট করেছে যা কিনা বলছে ইনপুট সঠিক হয় নি। এই লাইনগুলোর

## পাঠ ৭: স্ট্রিং অপারেশন

- স্ট্রিং তৈরি করা
- স্ট্রিং লেন্থ এবং স্ট্রিং অপারেশন
- ক্যারেকটার এক্সট্রাকশন
- স্ট্রিং কমপেরিজন
- স্ট্রিং সার্চিং এবং মডিফাইং
- ডাটা কনভারশন
- স্ট্রিং বাফার
- স্ট্রিং বিউন্ডার
- সারসংক্ষেপ

জাভাতে স্ট্রিং ব্যাপকভাবে ব্যবহৃত একটি অবজেক্ট। স্ট্রিং হচ্ছে কতগুলো ক্যারেক্টার-এর সিকুয়েন্স বা অনুক্রম। স্ট্রিং তৈরি করা খুব সহজ। যেমন –

```
String greeting = "Hello world!";
```

এখানে "Hello world!" হচ্ছে স্ট্রিং লিটারেল যা অকনেগুলো ক্যারেক্টার উদ্ধৃতি চিহ্নের ("" ) মাঝে লিখতে হয়।

জাভা কোডের মধ্যে কোন স্ট্রিং লিটারেল থাকলে কম্পাইলার সেটিকে String অবজেক্ট –এ পরিণত করে যার ড্যালু হয় উদ্ধৃতি চিহ্নের ("" ) মাঝের ক্যারেক্টার গুলো।

তবে অন্যান্য অবজেক্ট এর মতো String ও new কিওয়ার্ড এবং কন্সট্রাক্টর ব্যবহার করে তৈরি করতে পারি। String ক্লাসের ১৩ টি কনস্ট্রাক্টর আছে। সুতরাং আমরা আরও ১৩ টি উপায়ে স্ট্রিং তৈরি করতে পারি।

উদাহরণ –

```
String str = new String("Hello world!");
char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.' };
String helloString = new String(helloArray);
```

### String Length

String ক্লাসের মধ্যে length() মেথড থাকে যা একটি স্ট্রিং এর মধ্যে কতগুলো ক্যারেক্টার থাকে তার সংখ্যা রিটার্ন করে। String loremIpsum ="Lorem ipsum dolor sit amet.";

```
int len = loremIpsum.length();
```

### স্ট্রিং Concatenating

আমরা বেশ কয়েকটি উপায়ে স্ট্রিং কনকেট করতে পারি -

```
string1.concat(string2); // concat() মেথড ব্যবহার করে
"My name is ".concat("Rumplestiltskin"); // নিটোরেন ব্যবহার করে
"Hello," + " world" + "!" // + অপারেটর ব্যবহার করে
```

স্ট্রিং এর ভেতর বেশ কিছু মেথড আছে যেগুলো ব্যবহার করে আমরা স্ট্রিং মেনুপুলেট করতে পারি।

charAt() – এই মেথড ব্যবহার করে আমরা কোন ইন্ডেক্স এর ক্যারেক্টার আলাদা করতে পারি। উদাহরণ-

```
String hello = "Hello";
char getCharOfIndex2 = hello.charAt(2);
```

substring() – এই মেথড ব্যবহার করে আমরা একটি স্ট্রিং থেকে এর সাব-স্ট্রিং বা কোন নির্দিষ্ট অংশ আলাদা করতে পারি। উদাহরণ-

```
String str1 = "Hello world!";
String hello = str1.substring(0,5);
```

toLowerCase() – লোওয়ারকেস লেটারে কনভার্ট করার জন্যে এই মেথড ব্যবহার করি। toUpperCase() আপারকেস লেটারে কনভার্ট করার জন্যে এই মেথড ব্যবহার করি।

উদাহরণ –

```
String hello = "Hello";
hello.toUpperCase(); // HELLO
hello.toLowerCase(); // hello
```

নিচে আরও কিছু উদাহরণ দেখানো হল-

```
String str2 = "Hello world!";
int indexOfHaitch = str2.indexOf("H");
```

## বিশেষভাবে লক্ষণীয়

জাভাতে স্ট্রিং ক্লাস **immutable**, এর মানে হচ্ছে, একবার কোন স্ট্রিং অবজেক্ট তৈরি করলে তাকে আর পরিবর্তন করা যাবে না। আমরা অনেক ক্লাস লিখি, তারপর এর মাঝে বিভিন্ন ভ্যারিয়েবল রাখি, অবজেক্ট তৈরি করার পর সেই অবজেক্টের এর ভেতরের ভ্যারিয়েবল গুলো বিভিন্ন সময় পরিবর্তন করতে পারি। কিন্তু স্ট্রিং এর ক্ষেত্রে এটি সম্ভব নয়। অর্থাৎ আমরা যদি কোন একটি ভ্যালু দিয়ে একবার একটি স্ট্রিং অবজেক্ট তৈরি করি তাহলে সেটি আর পরিবর্তন করা যাবে না।

কিন্তু আমরা অনেকসময়ই স্ট্রিং কনকেট করি, সেক্ষেত্রে যা হয়, মনে করি-

```
String str = "Hello ";
str = str + "world";
```

এখানে যদিও মনে হচ্ছে আমরা স্ট্রিং এর ভ্যালু পরিবর্তন করে ফেলেছি। কিন্তু আসলে যা হচ্ছে তা হলো, আমরা প্রথমে একটি অবজেক্ট তৈরি করেছি, তারপর সেই অবজেক্ট এর ভ্যালু এবং নতুন একটি ভ্যালু নিয়ে নতুন একটি অবজেক্ট তৈরি করেছি, এবং যা str এখন নতুন সেই অবজেক্টকে রেফার করছে। আগের অবজেক্টটিকে গার্বজ কালেক্টর নিয়ে চলে যাবে।

এখন প্রশ্ন হচ্ছে, কেন এই **immutability** দরকার হয়।

স্ট্রিং পুল (**String Pool**) সম্পর্কে হয়তো অনেকেই জানি। এটি একটি জাভা হিপ এর একটি স্পেশাল এরিয়া। আমাদের যদি নতুন একটি স্ট্রিং তৈরি করতে হয়, সেই স্ট্রিং যদি আগে থেকেই স্ট্রিং পুল এ থেকে থাকে, তাহলে নতুন করে আর তৈরি না করে আগের অবজেক্টটির রেফারেন্স দেওয়া হয়। এতে করে আমাদের মেমরি ফুটপ্রিন্ট অনেক কমে যাচ্ছে।

```
String string1 = "abcd";
String string2 = "abcd";
```

আমরা যদি এই দুটি লাইন লিখি, তাহলে আসলে জাভা হিপ এ একটি স্ট্রিং অবজেক্ট-ই থাকবে, দুটি তৈরি হবে না। যদি স্ট্রিং **immutable** না হয়, তাহলে একটি স্ট্রিং যদি পরিবর্তন করি, তাহলে আসলে অন্যান্য রেফারেন্স গুলোও পরিবর্তন হয়ে যাবে।

এছাড়াও, আমরা জানি যে স্ট্রিং এর **hashcode** খুব বেশি ব্যবহার করা হয়। যেমন **HashMap**। স্ট্রিং **immutable** হওয়ায় এটা গ্যারান্টিড যে, সবসময় **hashcode** এক-ই হবে, সুতরাং আমরা প্রতিবার **hashcode** ক্যালকুলেট না করে নির্ধারিত ক্যাশিং করতে পারি।

আমরা স্ট্রিং প্যারামিটার হিসেব অনেক বেশি ব্যবহার করে থাকি, যেমন, নেটওয়ার্ক কানেকশন, ফাইল অপেনিং ইত্যাদির ক্ষেত্রে। সুতরাং এটি **immutable** না হলে পরিবর্তন করে ফেলা সম্ভব যা কিনা একটি সিরিয়াস রকম সিকিউরিটি থ্রেড হতে পারে। কিন্তু যেহেতু স্ট্রিং **immutable**, সুতরাং সেই সম্ভাবনা নেই।

তাছাড়া স্ট্রিং **immutable** হওয়ায় এটি ন্যাচারালি থ্রেড সেইফ, এবং স্বাধীনভাবে যে কেন থ্রেড একসেস করে পারে আমাদেরকে কষ্ট করে এর থ্রেড সেইফটি নিয়ে চিন্তা করতে হয় না।

## চলবে .....

## পাঠ ৮: জেনেরিকস

### জেনেরিকস ইন জাভা (Generics in Java)

আমরা জাভা-এর টাইপ সিস্টেম সম্পর্কে জানি। আমরা জানি জাভাতে কোন প্রোগ্রাম লিখতে হলে আমাদের কে টাইপ বলে দিতে হয়। যেমন আমরা যদি একটি মেথড লিখি তাহলে মেথডটি কি টাইপ প্যারামিটার এক্সপেক্ট করবে তা বলে দিতে হয়।

তবে জাভাতে একটি চমৎকার ফিচার আছে যাতে করে আমরা অনেক সময় টাইপ না বলে দিয়েই কোড লিখতে পারি। আমরা জেনেরিকস শুরু করার আগে একটি গুরুত্বপূর্ণ তথ্য জেনে নিই- জাভা প্রোগ্রামিং ল্যাংগুয়েজ এ সব ক্লাস **java.lang.Object** ক্লাসটিকে ইনহেরিট করে। আমরা এটি নিয়ে অন্য কোন চ্যাপ্টারে আলোচনা করবো, তবে এখন আমাদের শুধু এই তথ্যটুকু মনে রাখলেই চলবে।

সহজ কথায় যদি বলি, তাহলে জেনেরিকস দিয়ে আমরা যখন অবজেক্ট তৈরি করবো তখন টাইপ প্যারামিটারাইজ করতে পারি। অর্থাৎ আমরা যখন `new` অপারেটর দিয়ে অবজেক্ট তৈরি করবো তখন আসলে সিদ্ধান্ত নেবো এটির টাইপ কি হবে। এর আগে আমরা এমন ভাবে একটা ক্লাস বা মেথড লিখে ফেলতে পারি যাতে করে এটি যে কোন টাইপ এর জন্যে কাজ করে।

বরং একটা উদাহরণ দেখা যাক-

```
//একটি জেনেরিক ক্লাস , এখানে T হচ্ছে টাইপ প্যারামিটার যা অবজেক্ট তৈরি করার সময় রিয়েল টাইপ দিয়ে রিপ্লেস করা
public class Generic<T> {
    T obj;
    // একটা টাইপ অ্যারিয়ারন ডিক্লেয়ার করা হলো

    // কনস্ট্রাকটর - যে একটি রিয়েল অবজেক্ট অ্যাসাইনমেন্ট হিসেবে নেয়
    public Generic(T obj) {
        this.obj = obj;
    }

    // অবজেক্টটি এক্সেস করার জন্যে একটি মেথড
    public T getObj() {
        return obj;
    }

    // রানটাইমে অবজেক্ট-এর টাইপ জানেন কি , তা প্রিন্ট করে দেখি
    public void showType() {
        System.out.println("Type of T is: " + obj.getClass().getName());
    }

    public static void main(String[] args) {

        // একটি ইন্টিজার এর রেফারেন্স
        Generic<Integer> iObj;

        // অবজেক্ট তৈরি করি এবং iObj রেফারেন্স এ অ্যাসাইন করি এবং কনস্ট্রাকটর অ্যাসাইনমেন্ট হিসেবে 88 পাঠ্য করি
        iObj = new Generic<Integer>(88);

        // রানটাইমে এ তাহলে জেনেরিক ক্লাসটিতে T obj একটি ইন্টিজার হয়ে যাওয়ার কথা, প্রিন্ট করে দেখা যাক
        iObj.showType();

        int v = iObj.getObj();
        // ইন্টিজার অ্যানালুটি এর অ্যানালু এক্সেস করলে v তে রাখা হল

        System.out.println("value: " + v);
        // প্রিন্ট করি, যেখা যাক, আমরা এর অ্যানালু চিক ঠাক মতো পাওয়া যায় কিনা

        //এভাবে আমরা একটি স্ট্রিং টাইপ দিয়েও পরীক্ষা করতে পারি
        Generic<String> strObj = new Generic<String>("This is a Generics Test");
        strObj.showType();
        String str = strObj.getObj();
        System.out.println("value: " + str);
    }
}
```



এই প্রোগ্রামটি যদি রান করা হয়, তাহলে নিচের আউটপুট গুলো দেখা যাবে -

```
Type of T is: java.lang.Integer value: 88 Type of T is: java.lang.String value: This is a
Generics Test
```

আউটপুট গুলো থেকে বুঝা যাচ্ছে যে, আমাদের প্রোগ্রামটি সঠিক ভাবে কাজ করছে এবং একটি জেনেরিক ক্লাসে একটি ইন্টিজার এবং একটি স্ট্রিং প্যারামিটারাইজড করতে পেরেছি।

এভাবে আমরা আরও অন্যান্য টাইপ-ও প্যারামিটারাইজড করে পারি।

এবার আরও ভালভাবে এই প্রোগ্রামটি খেয়াল করা যাক-

```
public class Generic<T> {
    }
```

এখানে `T` হচ্ছে টাইপ প্যারামিটার। এটি মূলত একটি প্লেস হোল্ডার।

লক্ষ্য করুন – এর `T` কিন্তু `<>` এর মধ্যে থাকে।

আমরা সাধারণত যেভাবে ভ্যারিয়েবল ডিক্লেয়ার করি, সেভাবেই আমরা জেনেরিক্স-এ ভ্যারিয়েবল ডিক্লেয়ার করতে পারি। এর জন্যে আলাদা কোন নিয়ম নেই।

```
T obj;
```

এখানে `T` অবজেক্ট তৈরি করার সময় একটি রিয়েল অবজেক্ট অর্থাৎ আমরা যে অবজেক্ট প্যারামিটারাইজ করবো তা দ্বারা প্রতিস্থাপিত(replaced) হবে।

আমরা জানি যে জাভা একটি স্ট্যাটিক টাইপ অর্থাৎ টাইপ সেইফ ল্যাংগুয়েজ। অর্থাৎ জাভা কোড কম্পাইল করার সময় এর টাইপ ইনফরমেশন ঠিক ঠাক আছে কিনা তা চেক করে নেয়।

অর্থাৎ -

```
Generic<Integer> iObj;
```

এখানে `iObj` একটি ইন্টিজার প্যারামিটারাইজড অবজেক্ট রেফারেন্স।

```
iObj = new Generic<Double>(88.0); // Error!
```

এখন অবজেক্ট তৈরি করার সময় যদি ডাবল প্যারামিটারাইজড করি এবং `iObj` তে এসাইন করি, তাহলে

```
Error:(24, 16) java: incompatible types: Generic<java.lang.Double> cannot be converted to
```

কম্পাইল করার সময় উপরের ইররটি দেখতে পাবো।

## জেনেরিকস শুধুমাত্র অবজেক্ট নিয়ে কাজ করে-

আমরা জানি যে, জাভা দুই ধরনের টাইপ সাপোর্ট করে- `PrimitiveType` এবং `ReferenceType`। জেনেরিকস শুধুমাত্র `ReferenceType` অর্থাৎ শুধু মাত্র অবজেক্ট নিয়ে কাজ রে।

তাই-

```
Generic<int> intObj = new Generic<int>(50);
```

এই স্ট্যাটমেন্ট টি ভ্যালিড নয়। অর্থাৎ প্রিমিটিভ টাই এর ক্ষেত্রে জেনেরিকস কাজ করবে না।

জেনেরিক ক্লাস এর সিনট্যাক্স-

```
class class-name<type-param-list > {}
```

জেনেরিক ক্লাস ইনস্টেনসিয়েট করার সিনটেক্স-

```
class-name<type-arg-list > var-name = new class-name<type-arg-list >(cons-arg-list);
```

আমরা চাইলে একাধিক জেনেরিক টাইপ প্যারামিটারাইজড করতে পারি।

এবার তাহলে দুটি টাইপ প্যারামিটার নিয়ে একটি উদাহরণ দেখা যাক-

```

public class Tuple<X, Y> {
    private X x;
    private Y y;

    public Tuple(X x, Y y) {
        this.x = x;
        this.y = y;
    }

    public X getX() {
        return x;
    }

    public Y getY() {
        return y;
    }

    public void showTypes() {
        System.out.println("Type of T is " +
            x.getClass().getName() + " and Value: " + x);
        System.out.println("Type of V is " +
            y.getClass().getName() + " and Value: " + y);
    }

    public static void main(String[] args) {
        Tuple<String, String> tuple = new Tuple<String, String>("Hello", "world");
        tuple.showTypes();

        Tuple<String, Integer> person = new Tuple<>("Rahim", 45);
        person.showTypes();
    }
}

```

এই প্রোগ্রামটি রান করলে নিচের আউটপুট-টি পাওয়া যাবে –

```

Type of T is java.lang.String and Value: Hello Type of V is java.lang.String and Value:
world Type of T is java.lang.String and Value: Rahim Type of V is java.lang.Integer and
Value: 45

```

একটি টাপলের মধ্যে আমরা চাইলে আরেকটি টাপল রাখে পারি - নিচের উদাহরণটি চমৎকার-

```

Tuple<String, Tuple<Integer, Integer>> tupleInsideTuple = new Tuple<String, Tuple<Int

```

তবে আমরা যদি জাভা ৭ অথবা ৮ ব্যবহার করি তাহলে উপরের লাইনটি সংক্ষিপ্তভাবে লিখতে পারি –

```
Tuple<String, Tuple<Integer, Integer>> tupleInsideTuple = new Tuple<>("Tuple", ne
```

জান্না ৭ এ একটি নতুন অপারেটর সংযুক্ত হয়েছে যাকে বলা হয় ডায়মন্ড অপারেটর। এটি ব্যবহার করে আমরা জেনেরিকস এ verbosity কিছুটা কমানো যায়। অর্থাৎ

```
Map<String, List<String>> anagrams = new HashMap<String, List<String>>();
```

এই স্ট্যাটমেন্ট-টি অনেকটাই বড়। এটি আমরা এভাবে লিখতে পারি –

```
Map<String, List<String>> anagrams = new HashMap<>();
```

অর্থাৎ জেনেরিকস লেখার সময় বাম পাশে টাইপ প্যারামিটার ইনফরমেশন গুলো লিখলে ডান পাশে লিখতে হয় না। এটি অটোম্যাটিক্যালী ইনফার করতে পারে।

## Bounded Types

আমরা উপরে দুটি উদাহরণ দেখেছি যেগুলোতে আমরা যে কোন ধরনের টাইপ প্যারামিটারাইউজড করতে পারি। কিন্তু কখনো কখনো আমাদের টাইপ restrict করতে হয়। যেমন- আমরা একটি জেনেরিক ক্লাস লিখতে চাই যা কিনা একটি এর-তে রাখা কতগুলো নাম্বার-এর গড়(average) রিটার্ন করবে এবং আমরা চাই, এই এর-তে যে কোন ধরনের নাম্বার থাকতে পারে, যেমন- ইন্টিজার, ফ্লোটিং পয়েন্ট, ডাবল ইত্যাদি। আমরা টাইপ প্যারামিটার দিয়ে বলে দিতে চাই কখন কোনটা থাকবে। উদাহরণ দেখা যাক-

```
public class Stats<T> {
    T[] nums;

    public Stats(T[] nums) {
        this.nums = nums;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;
        for (T num : nums) {
            sum += num.doubleValue(); // Error!!!
        }

        return sum / nums.length;
    }
}
```

এভাবেজ ক্যালকুলেট করার জন্য আমাদের এভাবেজ মেথড সবসময় এরে থেকে ডাবল ভ্যালু এক্সপেক্ট করে। কিন্তু আমাদের এরে-এর টাইপ যেহেতু যে কোন রকম হতে পারে, সুতরাং সব অবজেক্ট থেকে ডাবল ভ্যালু পাওয়ার উপায় নেই।

ইনফ্যাক্ট এই ক্লাসটি কিন্তু কম্পাইল হবে না।

এই ক্লাসটিতে আমরা একটি restriction এড করতে পারি যাতে করে এই টাইপ প্যারামিটার শুধুমাত্র নাম্বার(ইন্টিজার, ফ্লোটিং পয়েন্ট, ডাবল) হবে, নতুবা এটি কাজ করবে না।

আমরা জানি যে সব নিউমেরিক অবজেক্ট গুলোর সুপার ক্লাস হচ্ছে `Number` . এবং `Number` এ `doubleValue()` মেথড ডিফাইন করা আছে। সুতরাং আমাদের ক্লাসটিকে একটু পরিবর্তন করি।

```
public class Stats<T extends Number> {
    T[] nums;

    public Stats(T[] nums) {
        this.nums = nums;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;
        for (T num : nums) {
            sum += num.doubleValue(); // Error!!!
        }

        return sum / nums.length;
    }
}
```

একটু লক্ষ্য করুন-

```
public class Stats<T extends Number>{
}
```

আমরা ক্লাস ডেফিনেশনে আমাদের টাইপ প্রেসহোল্ডার `T` নাম্বারকে extend করে। এটি আমাদের টাইপ প্যারামিটার পাস করতে restrict করে। অর্থাৎ আমরা শুধু মাত্র সেসব টাইপ পাস করতে পারবো যারা `Number` এর সাব টাইপ।

সুতরাং আমাদের এই `Stats` ক্লাস এখন `Integer`, `Double`, `Float`, `Long`, `Short`, `BigInteger`, `BigDecimal`, `Byte` ইত্যাদি অবজেক্ট এর জন্যে কাজ করবে।

সুতরাং দেখা যাচ্ছে যে, জেনেরিকস এর সুবিধা ব্যবহার করে আমরা এই স্ট্যাট ক্লাসটি আলাদা আলাদা করে অনেকগুলো না লিখে একটি দিয়েই কাজ করে ফেলা সম্ভব হল।

## Wildcard Arguments

নিচের উদাহরণটি লক্ষ্য করি-

```
ArrayList<Object> lst = new ArrayList<String>();
```

এটি যদি কম্পাইল করতে চেষ্টা করি, তাহলে কম্পাইলার incompatible types ইরর দেবে। কিন্তু আমরা জানি যে, সকল অবজেক্ট এর সুপার ক্লাস `Object`। তাছাড়া আমরা polymorphism থেকে জানি যে আমরা সাব ক্লাসের রেফারেন্স কে সুপার ক্লাসের রেফারেন্স এ এসাইন করতে পারি। সুতরাং উপরের স্ট্যাটমেন্ট-টি কাজ করার কথা।

নিচের উদাহরণ দুটি লক্ষ্য করি -

```
List<String> strLst = new ArrayList<String>(); // 1
List<Object> objLst = strLst; // 2 - Compilation Error
```

২ নম্বরের লাইনটি কাজ করছে না। যদিও বা এটি কাজ করে এবং আবিষ্কারি কোন একটি অবজেক্ট যদি `objLst` এড করা হয় তাহলে কিন্তু `strLst` করাপ্টেড হয়ে যাবে এবং সেটি আর স্ট্রিং থাকবে না।

ধরা যাক, আমরা একটা print মেথড লিখতে চাই যা কিনা একটি লিস্ট এর ইলিমেন্ট গুলো প্রিন্ট করে।

```
public static void print(List<Object> lst) { // accept List of Objects only,
    // not List of subclasses of object
    for (Object o : lst) {
        System.out.println(o);
    }
}
```

এটি কিন্তু শুধুমাত্র `List<Object>` একসেপ্ট করবে, `List<String>` অথবা `List<Integer>` করবে না।

উদাহরণ-

```
public static void main(String[] args) {
    List<Object> objLst = new ArrayList<Object>();
    objLst.add(new Integer(55));
    printList(objLst); // matches

    List<String> strLst = new ArrayList<String>();
    strLst.add("one");
    printList(strLst); // compilation error
}
```

এই সমস্যা দূর করার জন্যে জাভাতে একটি একটি অপারেটর ব্যবহার করা হয় – যার নাম wildcard (?)।

আমরা যদি আমাদের `print()` মেথডটি নিচের মতো করে লিখি, তাহলে কিন্তু আমাদের সমস্যা দূর হয়ে যাবে।

```
public static void print(List<?> lst) { // accept List of Objects only,
    // not List of subclasses of object
    for (Object o : lst) {
        System.out.println(o);
    }
}
```

`List<?> lst` এর মানে হচ্ছে আমরা এর টাইপ আমাদের জানা নেই, এটি যে কোন টাইপ হতে পারে। যেহেতু সব টাইপ এর সুপার ক্লাস `Object` সুতরাং এটি যেকোন টাইপ এর জন্যে কাজ করবে।

Bounded Types এর মতো আমরা Wildcard Arguments কেও Bounded করে ফেলতে পারি।

উদাহরণ -

```
public static void process(List<? extends Foo> list) { /* ... */ }
```

এটি শুধু মাত্র `Foo` এর সাব ক্লাস গুলো কে প্রসেস করতে পারবে। একে Upper Bounded Wildcards বলে।

আমরা যদি এমন কোন মেথড লিখতে চাই যা শুধু মাত্র `Integer`, `Number`, and `Object` প্রসেস করবে অর্থাৎ `Integer` এবং এর সুপার ক্লাস প্রসেস করবে তাহলে -

```
public static void addNumbers(List<? super Integer> list) {
}
```

একে Lower Bounded Wildcards বলে।

## Generic Methods

আমরা মূলত এতোক্ষণ জেনেরিক ক্লাস নিয়ে কথা বলেছি। আমরা একটি ক্লাসকে জেনেরিক না করে শুধুমাত্র এর একটি বা একাধিক মেথড কে জেনেরিক করে লিখতে পারি।

উদাহরণ-

```
public class Util {
    // Generic static method
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
            p1.getValue().equals(p2.getValue());
    }
}
```

এটি একটি জেনেরিক মেথড।

জেনেরিক মেথড-এ রিটার্নটাইপ এর আগে টাইপ-প্লেস হোল্ডার `<>` লিখতে হয়।

আমরা এবার চেষ্টা করবো কিভাবে আমরা একটি জেনেরিক সিংগলি লিংকলিস্ট লিখতে পারি --

```
/**
 * @author Bazlur Rahman Rokon
 * @date 2/4/15.
 */
public class SinglyLinkedList<Type> {
    private long size;

    private Node<Type> head;
    private Node<Type> tail;

    public void addFirst(Type value) {
        addFirst(new Node<>(value));
    }

    public void addLast(Type value) {
        addLast(new Node<>(value));
    }

    private void addLast(Node<Type> node) {
        if (size == 0) {
            head = node;
        } else {
            tail.setNext(node);
        }
        tail = node;
        size++;
    }

    public void addFirst(Node<Type> node) {
        Node<Type> temp = head;
        head = node;
        head.setNext(temp);

        size++;

        if (size == 1) {
            tail = head;
        }
    }

    public Node<Type> getHead() {
        return head;
    }

    public Node<Type> getTail() {
        return tail;
    }
}
```



```

public void removeFirst() {
    if (size != 0) {
        head = head.getNext();
        size--;
    }

    if (size == 0) {
        tail = null;
    }
}

public void removeLast() {
    if (size != 0) {
        if (size == 1) {
            head = null;
            tail = null;
        } else {
            Node<Type> current = head;

            while (current.getNext() != tail) {
                current = current.getNext();
            }
            current.setNext(null);
            tail = current;
        }
        size--;
    }
}

public Type getFirst() {
    return getHead().getValue();
}

// four scenario
// 1. empty list- do nothing
// 2. single node : ( previous is null)
// 3. Many nodes
//     a. node to remove is first node
//     b. node to remove is the middle or last

public boolean remove(Type type) {
    Node<Type> prev = null;
    Node<Type> current = head;

    while (current != null) {
        if (current.getValue().equals(type)) {
            if (prev != null) {
                // just skip the current node. it works fine
                prev.setNext(current.getNext());
            }
        }
        prev = current;
        current = current.getNext();
    }
    size--;
}

```

```

        if (current.getNext() == null) {
            tail = prev;
        }

        size--;
    } else {
        removeFirst();
    }

    return true;
}

prev = current;
current = current.getNext();
}

return false;
}

public long getSize() {

    return size;
}

public void print() {
    System.out.print("Total elements : " + size + " -> ");
    Node node = head;
    while (node != null) {
        System.out.print(node.getValue().toString() + " ,");
        node = node.getNext();
    }
    System.out.println();
}

public void clear() {
    for (Node<Type> x = head; x != null; ) {
        Node<Type> next = x.next;
        x.next = null;
        x.value = null;
        x = next;
    }

    head = tail = null;
    size = 0;
}

private class Node<Type> {
    private Type value;
    private Node<Type> next;
}

```

```
public Node(Type value) {  
    this.value = value;  
}  
  
public Type getValue() {  
    return value;  
}  
  
public void setValue(Type value) {  
    this.value = value;  
}  
  
public Node<Type> getNext() {  
    return next;  
}  
  
public void setNext(Node<Type> next) {  
    this.next = next;  
}  
}
```

এবার আমরা এটিকে রান করে দেখি-

```

/**
 * @author Bazlur Rahman Rokon
 * @date 2/4/15.
 */
public class LinkedListDemo {
    public static void main(String[] args) {
        SinglyLinkedList<Integer> integers = new SinglyLinkedList<>();
        integers.addFirst(4);
        integers.addFirst(3);
        integers.addFirst(2);
        integers.addFirst(1);

        integers.print();

        System.out.println("Remove first and last elements..");
        integers.removeFirst();
        integers.removeLast();
        integers.print();

        System.out.println("add elements at last ");
        integers.addLast(5);
        integers.addLast(6);
        integers.addLast(7);
        integers.print();

        SinglyLinkedList<String> stringLinkedList = new SinglyLinkedList<>();
        stringLinkedList.addFirst("abcd");
        stringLinkedList.addFirst("efgh");
        stringLinkedList.addFirst("ijkl");
        stringLinkedList.addFirst("mnop");
        stringLinkedList.addFirst("qrst");
        stringLinkedList.print();
    }
}

```

### Output:

Total elements : 4 - 1 ,2 ,3 ,4 , Remove first and last elements.. Total elements : 2 - 2 ,3 , add elements at last Total elements : 5 - 2 ,3 ,5 ,6 ,7 , Total elements : 5 - qrst ,mnop ,ijkl ,efgh ,abcd ,

## পাঠ ৯: জাভা আই/ও

- স্ট্রিম
- বাইট স্ট্রিম
- ক্যারেক্টার স্ট্রিম
- বাফারড স্ট্রিম
- স্ক্যানিং এবং ফরমেটিং
- ডাটা স্ট্রিম
- ইনপুট স্ট্রিম
- আউটপুট স্ট্রিম
- ফাইল
- রিডিং এ টেক্সট ফাইল
- রাইটিং এ টেক্সট ফাইল
- সারসংক্ষেপ

ইনপুট আউটপুট সংক্ষেপে যাকে আমরা বলি আই/ও (I/O) যে কোন কম্পিউটার সিস্টেম বা প্রোগ্রামিং ল্যাংগুজের একটি মৌলিক বিষয়। যে কোন প্রোগ্রাম লিখতে গেলেই আসলে আমাদের আই/ও দরকার হয়। তবে এই বিষয়টি ঠিক ততটা মজার না যতটা অন্যান্য বিষয় গুলো। খানিকটা ইলেক্ট্রিসিটির মতো। আমরা জানি প্রত্যেকটি বাড়িতেই এটি আছে, দরজা দিয়ে প্রবেশ করেই আমাদের হাত সুইচবোর্ডের দিকে চলে যায়, আমার সুইচ টিপ দিই, এবং লাইট জ্বলে উঠে। এর পেছনের ব্যপারগুলো নিয়ে যেমন ইলেক্ট্রিসিটি কোথা থেকে এলো, কিভাবে কাজ করে এসব নিয়ে আমাদের চিন্তা করতে হয় না। এগুলো নেপথ্যে থেকে ঠিক ঠাক মতো কাজ করে। আই/ও অনেকটা এরকম।

এবার ইনপুট আউটপুটকে সংজ্ঞায়িত করা যাক। একটি প্রোগ্রাম মূলত ডাটা আর ফাংশন এর সমষ্টি। অর্থাৎ ফাংশন ডাটা গুলো নিয়ে কাজ করে। তো এই ডাটা গুলো কোথাও থেকে তৈরি হয় এবং সেগুলোকে আমাদের প্রোগ্রাম ফাংশন প্রসেস করে। প্রসেসকৃত ডাটা গুলো হচ্ছে আউটপুট। সহজ করে বলা যেতে পারে, আমাদের প্রোগ্রাম কোন সোর্স থেকে ডাটা পড়ে এবং কোন একটা ডেস্টিনেশনে রাইট করে। উদাহরণ হিসেবে দেওয়া যেতে পারে- আমাদের কিবোর্ড একটি ডাটা সোর্স। আমরা একটা প্রোগ্রাম লিখতে পারি যা কি বোর্ড এ ডাটা টাইপ করছি তা ইনপুট হিসেবে নিচ্ছে এবং `System.out.println()` মেথড দিয়ে সেগুলো কনসোলে প্রিন্ট করতে পারি।

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class StandardIOExample {
    public static void main(String[] args) throws IOException {
        BufferedReader reader;
        reader = new BufferedReader(new InputStreamReader(System.in));
        String line;
        do {
            line = reader.readLine();
            line = line.toUpperCase();
            System.out.println(line);
        } while (!line.equals("quit"));
    }
}
```

উপরের প্রোগ্রামটি কিবোর্ড থেকে একটি লাইন পড়ে সেটি আপারকেইস এ কনভার্ট করে কনসোলে প্রিন্ট করে। একটি একটি সরলতম এবং খুবই প্রয়োজনীয় ইনপুট/আউটপুট এর উদাহরণ। সাধারণত আমরা কোন একটি ফাইল থেকে ডাটা পড়ি এবং প্রয়োজনীয় প্রসেসিং এর পর অন্য একটি ফাইল এ রাইট করি। তবে ইনপুট আউটপুট শুধুমাত্র ফাইল এর মধ্যে সীমাবদ্ধ থাকবে এমন কোন কথা নেই। আমরা চাইলে একটা স্ট্রিং অবজেক্ট থেকে ডাটা পড়ে আরেকটি স্ট্রিং অবজেক্ট রাইট করতে পারি। এক্ষেত্রে ইনপুট হচ্ছে একটি স্ট্রিং অবজেক্ট এবং আউটপুটও একটি স্ট্রিং অবজেক্ট। আবার একটি ফাইল থেকে ডাটা পড়ে একটি স্ট্রিং অবজেক্ট এ রাখতে পারি। এভাবে অনেক গুলো কন্সট্রিকশন করতে পারি। তবে সব সময় যে ইনপুট এবং আউটপুট এক সাথেই কাজ করতে হবে এমনটা নয়। কখনো কখনো শুধুমাত্র ইনপুট অথবা শুধুমাত্র আউটপুট নিয়ে একটি প্রোগ্রাম তৈরি হতে পারে।

তবে একজন জাভা প্রোগ্রামার এর কাছে আই/ও অনেক গুলো কারণেই গুরুত্বপূর্ণ হতে পারে। জাভাতে অনেক গুলো আই/ও ক্লাস এর কোর এপিআই এর সাথেই থাকে যার বেশির ভাগ – java.io প্যাকেজ-এ। তবে জাভাতে অধিকাংশ ক্ষেত্রেই আই/ও দুই ভাগে ভাগ করা হয়েছে। একটি হলো বাইট ভিত্তিক আই/ও যা input stream এবং output stream দিয়ে হ্যান্ডেল করা হয়, এবং অন্যটি হলো ক্যারেকটার ভিত্তিক যা readers এবং writers দিয়ে হ্যান্ডেল করা হয়। তবে দুই টাইপ-এ অ্যাবস্ট্রাকশন সরবরাহ করে যা দিয়ে সোর্সের সঠিক টাইপ না জেনেও পড়তে বা লিখতে পারি। এতে করে আমরা একি মেথড দিয়ে কনসোল থেকে ডাটা পড়তে পারছি আবার সেই মেথড দিয়ে আমরা নেটওয়ার্ক কানেকশন থেকেও পড়তে পারছি। এতো হল টিপ অব দি আইসবার্গ। একবার আমরা অ্যাবস্ট্রাকশন এ অভ্যস্ত হয়ে গেলে যে কোন সোর্স থেকে ডাটা পড়তে পারবো, আমাদের আসলে খুব একটা কেয়ার করতে হবে না কিভাবে বা কোন সোর্স থেকে ডাটা আসছে বা যাচ্ছে। এখানে একটা গুরুত্বপূর্ণ কথা বলে রাখি, সেটা হলো, জাভা প্রোগ্রামারদের সব থেকে পছন্দের বিষয় হচ্ছে অ্যাবস্ট্রাকশন। অনেক ভূমিকা হলো, এবার তাহলে আরো ভেতরে প্রবেশ করা যাক। শুরুতেই ফাইল নিয়ে কাজ করা যাক।

### ওয়ার্কিং উয়িদ ফাইল

পাথ প্রত্যেকটি ফাইল এর জন্যে একটি নির্দিষ্ট পাথ থাকে যাতে করে আমরা আলাদা করতে পারি। পাথ হচ্ছে কতগুলো ক্যারেকটার এর সমষ্টি এবং এতে ফাইলে এর নাম এবং ডিরেকটরী লোকেশন থাকে। যেমন ওয়িন্ডোজ প্লাটফর্মের ক্ষেত্রে C:\users\rokonoid\hello.txt হচ্ছে hello.txt ফাইল এর পাথনেইম যা কিনা C ড্রাইভের users ডিরেকটরির মাঝে rokonoid ডিরেকটরিতে আছে। Unix প্লাটফর্মের ক্ষেত্রে /home/rokonoid/hello.txt হচ্ছে hello.txt এর পাথনেইম।

পাথনেইম দুই প্রকার হতে পারে- absolute path এবং relative path. Current working directory বলে একটা কনসেপ্ট আছে, আর সেটি হলো, আমরা যখন যে ডিরেকটরিতে কাজ করি। মনে করা যাক আমাদের জাভা প্রোগ্রামটি /home/rokonoid বা C:\users\rokonoid ডিরেকটরিতে আছে। তাহলে আমাদের কারেন্ট ওয়ার্কিং ডিরেকটরি হচ্ছে C:\users\rokonoid বা /home/rokonoid। এখন এই ডিরেকটরিতে যদি একটি hello.txt ফাইল থাকে, তাহলে এই ফাইল এর রিলেটিভ পাথ হবে hello.txt আর absolute path পাথ হবে C:\users\rokonoid\hello.txt বা /home/rokonoid/hello.txt। রিলেটিভ পাথ কারেন্ট ওয়ার্কিং ডিরেকটরি থেকে রিজলভ করা যায়।

### ফাইল তৈরি

এবার দেখা যাক কিভাবে একটি ফাইল অবজেক্ট তৈরি করা যায়। *java.io.File* ক্লাসটি একটি পাথ এর ফাইল বা ডিরেকটরিকে রিপ্রেজেন্ট করে। এ ক্লাসে বেশ কয়েকটি কনস্ট্রাক্টর রয়েছে, এর মানে বেশ কয়েক উপায়ে একটি ফাইল অবজেক্ট তৈরি করা যায়।

```
File(String pathname)
File(File parent, String child)
File(String parent, String child)
File(URI uri)
```

এখন আমাদের একটি পাথনেইম যদি হয় hello.txt বা /home/rokonoid/hello.txt তাহলে আমরা নিচের মতো করে ফাইল অবজেক্ট তৈরি করতে পারি।

```
File file = new File("hello.txt");
```

### অথবা

```
File file = new File("/home/rokonoid/hello.txt");
```

এই ফাইলটি আমাদের দেওয়া পাথ এ যে ফাইলটি আছে তাকে রিপ্রেজেন্ট করে। তবে মজার ব্যাপার হচ্ছে ফাইল অবজেক্ট তৈরি করতে হলে এই পাথটি ফিজিক্যালি থাকতে হবে এমন কোন কথা নেই। File ক্লাসের বেশি কিছু মেথড আছে যেগুলো দিয়ে আমরা দেখতে পারি এই ফাইলটি আসলেই আমরা যে পাথটি দিয়েছি সেখানে আছে কিনা। না থাকলে আমরা তৈরি করতে পারি।

```
import java.io.File;
import java.io.IOException;

public class FileExample {

    public static void main(String[] args) {
        File file = new File("hello.txt");
        if (file.exists()) {
            System.out.println("File exists");
        } else {
            System.out.println("File does not exist,lets create one");
            try {
                file.createNewFile();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

উপরের উদাহরণটিতে আমরা প্রথমে আমাদের দেওয়া পথ দিয়ে একটি ফাইল অবজেক্ট তৈরি করেছি। তারপর দেখেছি এই ফাইলটি আসলেই ফিজিক্যালি আমাদের দেওয়া পথ এ আছে কিনা। যদি না থাকে, তাহলে সেই পথ এ নতুন একটি ফাইল তৈরি করা হয়েছে।

এছাড়াও আরও কিছু বেশ প্রয়োজনীয় মেথড যেমন- `isFile()` এবং `isDirectory()` আছে যেগুলো দিয়ে আমরা বের করতে পারি কোন পথ ফাইল বা ডিরেকটরি কিনা।

এছাড়াও কারেন্ট ওয়ার্কিং ডিরেকটি বের করা জন্যে একটি বিশেষ উপায় হলো -

```
public class CurrentWorkingDirectory {
    public static void main(String[] args) {
        String workingDir = System.getProperty("user.dir");
        System.out.println(workingDir);
    }
}
```

### পাথ সেপারেটর

একটি বিষয় মনে রাখতে হবে যে বিভিন্ন প্ল্যাটফর্ম ফাইলের পাথ এর দুটি পার্ট আলাদা করার জন্যে আলাদা ক্যারেক্টার ব্যবহার করে থাকে। যেমন- windows ব্যাকস্লেশ (/) এবং unix সিস্টেম ফরওয়ার্ড স্লেশ (/) ব্যবহার করে থাকে। সুতরাং পাথ তৈরি করতে হলে খেয়াল রাখা জরুরী কোন প্ল্যাটফর্মে থেকে প্রোগ্রামটি রান করা হচ্ছে। কিন্তু আমাদের যেহেতু মূল উদ্দেশ্য প্ল্যাটফর্ম স্পেসিফিক কোড না লেখা, সেক্ষেত্রে নিজের উপায়টি ব্যবহার করা যেতে পারে।

```
String workingDir = System.getProperty("user.dir");
String newFile = workingDir + File.separator + "hellword.txt";
File file = new File(newFile);
```



এখানে File.separator একটি কনস্ট্যান্ট যা যে প্লাটফর্মে প্রোগ্রামটি রান করছে তার উপর ভিত্তি করে সেপারেটর স্ট্রিং আকারে দিয়ে থাকে।

### ডিরেকটরি তৈরি

File ক্লাসে এ mkdir() এবং mkdirs() দুটি মেথড আছে যেগুলো ব্যবহার করে আমরা একটি ডিরেকটরি তৈরি করতে পারি। এবং এদের মাঝে ফাইল তৈরি করতে পারি।

```
import java.io.File;
import java.io.IOException;

public class DirectoryExample {
    public static void main(String[] args) throws IOException {
        File dir = new File("/home/rokonoid/myDir");

        dir.mkdir();

        String dirPath = dir.getPath();
        System.out.println("Directory Path: " + dirPath);

        // lets create a new file
        String fileName = "hello.txt";
        File file = new File(dirPath + File.separator + fileName);
        file.createNewFile();

        String filePath = file.getPath();
        System.out.println("File Path: " + filePath);
    }
}
```

এই প্রোগ্রামটি রান করলে নিচের আউটপুট পাওয়া যাবে -

```
Directory Path: /home/rokonoid/myDir
File Path: /home/rokonoid/myDir/hello.txt
```

### ফাইল রিনেমিং, কপিং এবং ডিলেটিং

File ক্লাস এ renameTo() ব্যবহার করে আমরা ফাইল রিনেম করতে পারি।

```
import java.io.File;

public class FileRenameExample {

    public static void main(String[] args) {
        File oldFile = new File("old_hello.txt");
        File newFile = new File("new_hello.txt");

        boolean fileRenamed = oldFile.renameTo(newFile);

        if (fileRenamed) {
            System.out.println(oldFile + " renamed to " + newFile);
        } else {
            System.out.println("Renaming " + oldFile + " to " + newFile + " failed.");
        }
    }
}
```

ফাইল ডিলিট করার জন্যে দুটি মেথড রয়েছে- delete() এবং deleteOnExit() এই মেথড দুটি দিয়ে ফাইল এবং ডিরেকটরী ডিলেট করা যায়। তবে ডিরেকটরী ডিলিট করতে হলে অবশ্যই ডিরেকটরি টি খালি থাকতে হবে, অর্থাৎ ডিরেকটরীতে যদি আরও ফাইল থাকে, তাহলে সেগুলো আগে ডিলিট করে ফেলতে হবে। delete() মেথডটি সাথে সাথেই কাজ করে তবে, deleteOnExit() মেথডটি যখন JVM টারমিনেট করে তখন ডিলেট করে। আমাদের অনেকসময় প্রোগ্রাম চলাকালিন টেম্পোরারি ফাইল তৈরি করার দরকার পরে যা প্রোগ্রাম টার্মিনেট হয়ে গেলে দরকার হয় না, সেসব ক্ষেত্রে এই মেথড ব্যবহার করা যেতে পারে।

```
public class FileDeleteExample {
    public static void main(String[] args) {
        // To delete the hello.txt file immediately
        File file1 = new File("hello.txt");
        file1.delete();

        // To delete the hello.txt file when the JVM terminates
        File file2 = new File("hello.txt");
        file2.deleteOnExit();
    }
}
```

File ক্লাসে কোন মেথড নেই যাতে করে সরাসরি আমরা ফাইল কপি করতে পারি। একটি ফাইল কপি করতে হলে আমাদেরকে একটি নতুন ফাইল তৈরি করতে হবে এবং সেই ফাইল এর কন্টেন্ট গুলো রিড করে নতুন ফাইল এ রাইট করতে হবে। পরবর্তি চ্যাপ্টারে এ নিয়ে আলোচনা করা হবে। লিস্টিং ফাইলস

আমরা একটি ডিরেকটরিতে কতগুলো ফাইল আছে তার লিস্ট listFiles() মেথড দিয়ে সহজেই বের করে ফেলতে পারি। উদাহরণ-

```
import java.io.File;

public class ListingFiles {
    public static void main(String[] args) {
        File home = new File("/home/rokonoid/");

        File[] listRoots = home.listFiles();
        for (File file : listRoots) {
            System.out.println(file.getPath());
        }
    }
}
```

### ফাইল ফিল্টার

তবে অনেক সময় আমাদের ফাইল ফিল্টারের প্রয়োজন হয়। মনে করা যাক একটি ডিরেকটরীতে শুধুমাত্র png ফাইল গুলো আমাদের দরকার। সেক্ষেত্রে -

```
import java.io.File;
import java.io.FileFilter;

public class FileFilterExample {
    public static void main(String[] args) {
        File home = new File("/home/rokonoid/Pictures");

        FileFilter pngFilter = new FileFilter() {

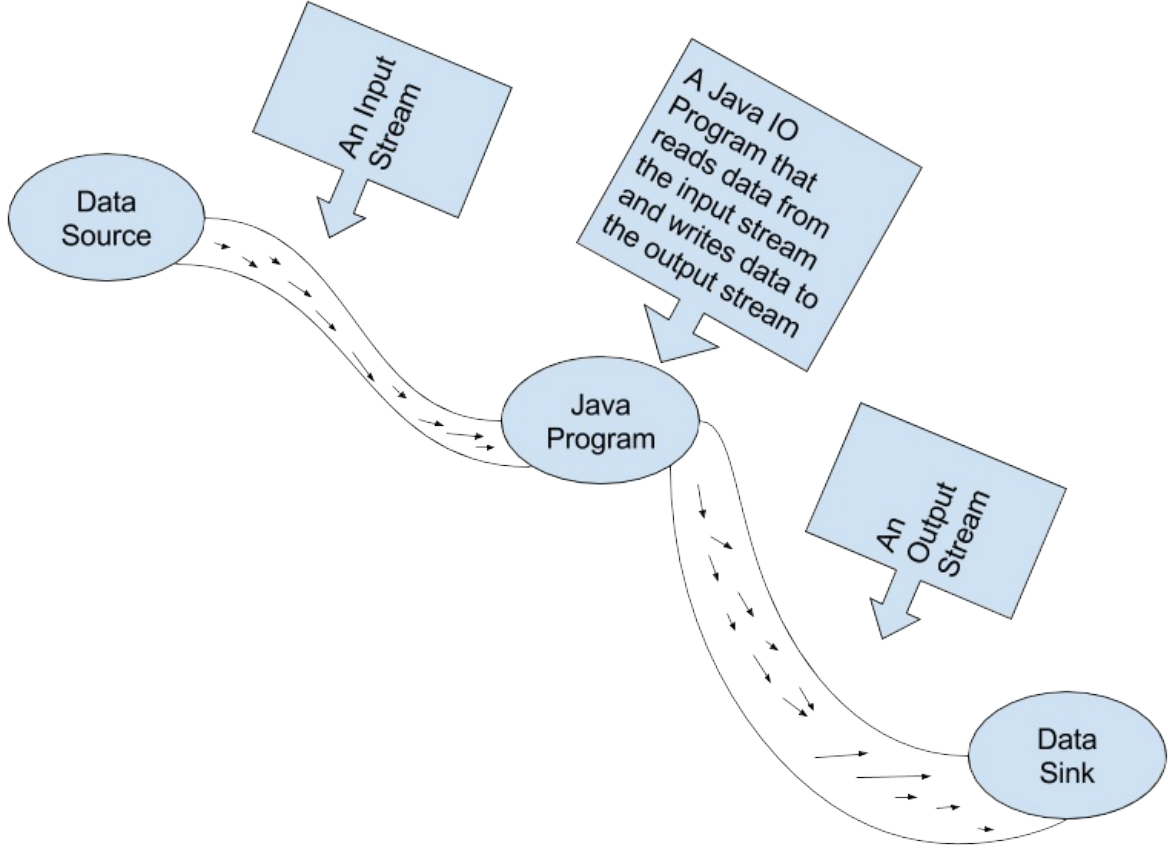
            @Override
            public boolean accept(File pathname) {
                String fileName = pathname.getName();
                if (fileName.endsWith(".png")) {
                    return true;
                }
                return false;
            }
        };

        File[] listRoots = home.listFiles(pngFilter);
        for (File file : listRoots) {
            System.out.println(file.getPath());
        }
    }
}
```

উপরের উদাহরণটিতে FileFilter এর একটি anonymous ক্লাস লেখা হয়েছে যা কিনা listFiles() মেথডটি parameter হিসেবে নিচ্ছে। এই ফিল্টারের accept() মেথডটিতে আমরা আমাদের ফিল্টার লজিকটুকু লেখা হয়েছে যাতে করে এটি শুধুমাত্র png ফাইল গুলো লিস্টিং করে।

### ইনপুট/আউটপুট স্ট্রিম

স্ট্রিম এর আক্ষরিক অর্থ হচ্ছে প্রবাহ । এর মানে হচ্ছে অনেকটা পানির ধারার মতো একটি উৎস থেকে অবিরাম ভাবে প্রবাহ হচ্ছে এমন কিন্তু আমরা ঠিক ভাবে উৎসে কতটুকু পানি আছে জানি না । অর্থাৎ কনসেপচুয়ালি একটি অবিরাম ডাটা প্রবাহ । আমরা এই প্রবাহ থেকে ডাটা পড়তে বা লিখতে পারি । যে কোন স্ট্রিম একটি উৎস বা গন্তব্যস্থলের সাথে সংযুক্ত । উৎস কে বলা হয় ডাটা সোর্স এবং গন্তব্যস্থলকে বলা হয় ডাটা সিংক ।



### ইনপুট স্ট্রিম তৈরি

ছবিতে দেখা যাচ্ছে একটি সোর্স থেকে প্রবাহ আকারে ডাটা ফ্লো হচ্ছে জাভা প্রোগ্রামে । এবং জাভা প্রোগ্রামটি আরেকটি ডাটা ফ্লো তৈরি করছে যা গন্তব্যে পৌঁছাচ্ছে । তাহলে একটি সোর্স থেকে ডাটা পড়তে হলে আমাদেরকে কয়েকটি ধাপে যেতে হয় - ১. প্রথমে একটি সোর্স নির্ধারণ করতে হবে । সোর্স একটি স্ট্রিং হতে পারে, কিংবা একটি ফাইল অথবা একটি নেটওয়ার্ক কানেকশন । ২. সোর্স এর উপর ভিত্তি করে একটি ইনপুট স্ট্রিম তৈরি করতে হবে । ৩. ইনপুট স্ট্রিম থেকে ডাটা পড়া । সাধারণত একটু লুপ এর মধ্যে ইনপুট স্ট্রিম এর `read()` মেথড কল করতে হয় , এবং লুপটি ততক্ষণ পর্যন্ত চলে যতক্ষণ পর্যন্ত ডাটা পড়া শেষ না হয় ।

### ইনপুট স্ট্রিম থেকে ডাটা পড়া

স্ট্রিম দুই প্রকার হতে পারে-

1. বাইট স্ট্রিম
2. ক্যারেকটার স্ট্রিম । বাইট স্ট্রিম

বাইট ভিত্তিক আইও নিয়ে কাজ করার জন্যে বাইট স্ট্রিম-এ বেশ সমৃদ্ধ ক্লাস আছে। সাধারণত বাইট স্ট্রিম যে কোন টাইপ অবজেক্ট (যেমন বাইনারী ডাটা) তে ব্যবহার করা যায়। সব বাইট স্ট্রিম এর ক্লাস গুলো `InputStream` এবং `OutputStream` এর সাব ক্লাস। যদিও আরও অনেক বাইট স্ট্রিম ক্লাস আছে, কিন্তু যেহেতু এই দুটি ক্লাস সবার উপরে, আমরা শুরুতেই এই দুটি ক্লাস নিয়েই কথা বলবো।

`java.io.InputStream` এটি একটি অ্যাবস্ট্রাক্ট ক্লাস এবং সকল ইনপুট স্ট্রিম এর সুপার ক্লাস। এতে তিনটি বেসিক মেথড আছে যা কিনা কিভাবে ডাটা স্ট্রিম থেকে পড়তে হয় তা নিয়ে ডিল করে। এছাড়াও স্ট্রিম ক্লোজ করা, ফ্লাস করা, এবং কতগুলো বাইট আরও পড়তে হবে ইত্যাদি নিয়ে কিছু মেথড আছে। এগুলো নিয়ে একটি ডিটেইল ব্যাংখ্যা করা যাক। `read()` মেথড:

```
public abstract int read() throws IOException
```

এই মেথডটি ১ বাইট unsigned ডাটা পড়ে এবং এর ইন্টিজার ভ্যালু রিটার্ন করে যা কি না ০ থেকে 255 এর মধ্যে। যদি কোন বাইট না পাওয়া যায় তাহলে এটি -1 রিটার্ন করে এবং এতে করে আমরা বুঝতে পারি স্ট্রিম এর ডাটা শেষ হয় গেছে। আমরা একটি উদাহরণ দেখি। যেহেতু ইনপুট স্ট্রিম একটি অ্যাবস্ট্রাক্ট ক্লাস এবং এর বেশ কিছু সাব ক্লাস আছে, উদাহরণ দেওয়ার সুবিধার্থে আমরা একটি ফাইল ইনপুট স্ট্রিম ব্যবহার করি যা কিনা কোন একটি লোকেশানে রাখা একটি টেক্সট ফাইল পড়তে পারবে। প্রথমে আমরা একটি টেক্সট ফাইল তৈরি করে কোন একটি লোকেশানে রাখি। সাধারণত প্রজেক্ট এর একটি ফোল্ডার তৈরি করে তাতেও রাখা যেতে পারে। এর পর এই ফাইল এ যে কোন একটি স্ট্রিং লিখি। এখানে আমার ফাইল এর নাম `input.txt` এতে নিচের লাইটি লিখেছি - `The quick brown fox jumps over the lazy dog.` এবার নিচের কোডটি রান করি।

```

import java.io.FileInputStream;
import java.io.IOException;

public class InputStreamExample {
    public static void main(String[] args) {
        FileInputStream in = null;
        try {
            in = new FileInputStream("input.txt");
            int c;

            while ((c = in.read()) != -1) {
                System.out.print(c + ",");
            }
        } catch (IOException e) {
            System.err.println("Could not read file");
        } finally {
            if (in != null) {
                try {
                    in.close();
                } catch (IOException e1) {
                    System.err.println("Could close input stream");
                }
            }
        }
    }
}

```

এখানে শুরুতে একটি `FileInputStream` ক্লাস এর ইনস্ট্যান্স ক্রিয়েট করা হয়েছে। যেহেতু `InputStream` একটি abstract ক্লাস, এবং আমাদের ডাটা সোর্স একটি ফাইল, সুতরাং কংক্রিট ক্লাস হিসেবে `FileInputStream` ব্যবহার করা হয়েছে। এতে আর্গুমেন্ট হিসেবে আমাদের টেক্সট ফাইলটির লোকেশান দেয়া হয়েছে। এখানে এটি রিলেটিভ পাথ। আমাদের ওয়ার্কিং ডিরেকটরী হচ্ছে প্রজেক্ট ডিরেকটরী, যেহেতু ফাইলটি প্রজেক্ট ডিরেকটরীতেই রাখা আছে। যদি ফাইলটি অন্য ডিরেকটরীতে থাকে সেক্ষেত্রে absolute পাথ দিতে হবে।

তারপর একটা `int c` ডিক্লেয়ার করা হয়েছে। এরপর একটি লুপ রয়েছে। এতে প্রতিবার একটি করে বাইট রিড করে `c` তে এসাইন করা হচ্ছে এবং তা প্রিন্ট আউট করা হচ্ছে। এই লুপটি ততক্ষণ পর্যন্ত চলবে যতক্ষণ পর্যন্ত `read()` মেথডটি -1 রিটার্ন না করে। ফাইলটি পড়া শেষ হয়ে গেলে এটি -1 রিটার্ন করবে। কোডটি একটি ট্রাই ক্যাচ ব্লক এর মধ্যে কারণ আমার জানি যে আই/ও আছে খুব লো-লেভেল থেকে কাজ করে। এর মাঝে কোন একটি সমস্যা হতেই পারে এবং তা হলে JVM `IOException` ত্রুটি করবে এবং তা যাতে আমরা হ্যান্ডেল করতে পারি। এছাড়াও একটি ফাইনালি ব্লক আছে যেখানে আমরা স্ট্রিমটি বন্ধ করেছি। আমাদের খেয়াল রাখতে হবে যে, যখনি একটি স্ট্রিম এর কাজ শেষ হয়ে যাবে তখনি তা বন্ধ করে দিতে হবে। এটি অনেকটা আমাদের ওয়াশরুমের পানির টেপ এর মতো। কাজ শেষ হলে আমরা অফ করে দিই যাতে করে রিসোর্স নষ্ট না হয়।

এখন উপরের কোডটি যদি রান করি তাহলে কনসোলে আমরা নিচের আউটপুটটি দেখতে পাবো-

```

84,104,101,32,113,117,105,99,107,32,98,114,111,119,110,32,102,111,120,32,106,117,109,1
12,115,32,111,118,101,114,32,116,104,101,32,108,97,122,121,32,100,111,103,46,

```

এর কারণ হচ্ছে read() মেথডটি এক সাথে একটি বাইট পড়ে এবং এর ইন্টিজার রিপ্রেজেন্টেশন রিটার্ন করে। আমরা যদি একে ঠিক আমাদের input.txt এর স্ট্রিং এর মতো করে প্রিন্ট করতে চাই তাহলে ইন্টিজারকে ক্যারেকটার এ কাস্ট করতে হবে। System.out.print((char)c);

### আউটপুট স্ট্রিম তৈরি

ছবিতে দেখা যাচ্ছে যে জাভা প্রোগ্রামটি একটি আউটপুট স্ট্রিম ব্যবহার করে একটি ডাটা সিংক ডাটা ট্রান্সফার করেছে। আউটপুট স্ট্রিমের মাধ্যমে প্রোগ্রাম থেকে ডাটা ডাটা সিংকে পাঠাতে হলে কয়েকটি ধাপ-এ যেতে হয়- ১. প্রথমে একটি ডাটা সিংক নির্ধারণ করতে হবে। এটি একটি ফাইল হতে পারে, কিংবা একটি স্ট্রিং অবজেক্ট বা নেটওয়ার্ক কানেকশন। ২. ডাটা সিংক ব্যবহার করে একটি আউটপুট স্ট্রিম অবজেক্ট তৈরি করতে হবে। ৩. এরপর আউটপুট স্ট্রিমটি ফ্লাস করতে হবে। ৪. এবং সবশেষে আউটপুট স্ট্রিমটি ক্লোজ করে দিতে হবে।

### আউটপুট স্ট্রিমে ডাটা রাইট করা

এবার আমরা চেষ্টা করবো ডাটা কিভাবে ডাটা সিংকে রাইট করা যায়। এক্ষেত্রে ডাটা সিংক হিসেবে একটি ফাইল নিতে পারি। আউটপুট স্ট্রিম হিসেবে নিতে পারি FileOutputStream. OutputStream এর একটি একটি মেথড হচ্ছে write() যা দিয়ে আমরা ডাটা ফাইল এ রাইট করতে পারি। write() মেথড এর কগুলো অভ্যন্তরীণ আছে। এর যেকোন একটা ব্যবহার করতে পারি। একটি স্ট্রিং অবজেক্ট থেকে আমরা সহজেই ডাটা বাইট আকারে একটি অ্যারেতে রাখতে পারি।

```
String text = "Hello";
byte[] textBytes = text.getBytes();
```

এরপর এই বাইট অ্যারেকে আউটপুট স্ট্রিম এর আরইট মেথডে আর্গুমেন্ট হিসেবে পাস করতে পারি। উদাহরণ-

```
import java.io.FileOutputStream;
import java.io.IOException;

public class OutputStreamExample {
    public static void main(String[] args) {
        String destFile = "output.txt";
        String data = "Lorem ipsum dolor sit amet," +
            " consectetur adipiscing elit. " +
            " Suspendisse at placerat ipsum. ";
        try {
            FileOutputStream fos = new FileOutputStream(destFile);
            fos.write(data.getBytes());
            fos.flush();
            fos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

এরপর আউটপুট স্ট্রিমটিকে ফ্লাশ করতে হয় flush() মেথড ব্যবহার করে। আমাদের উদ্দেশ্য হচ্ছে ডাটা সিংকে ডাটা রাইট করা। এক্ষেত্রে আমরা FileOutputStream এ ডাটা রাইট করছি যা কিনা একটি ফাইল এর অ্যাবস্ট্রাকশন। আউটপুট স্ট্রিম বাইট গুলোকে অপারেটিং সিস্টেম কে দেয় যে কিনা আসলে বাইট গুলো ফাইল এ রাইট করার জন্যে রেসপনসিবল। অপারেটিং সিস্টেম আসলে নির্ধারণ করে কখন বাইট গুলো ফাইল এ রাইট করবে কিন্তু আমাদের আগে সবগুলো বাইট অপারেটিং সিস্টেমকে দিতে হবে। আউটপুট স্ট্রিম এর যেহেতু অ্যাবস্ট্রাক্ট ক্লাস এবং এর অনেক গুলো কনক্রিট আছে, এদের কোন কোন ক্লাস নিজের মাঝে বাইট গুলোর বাফার রেখে দিতে পারে। এক্ষেত্রে flush() মেথডটি বাফার ক্লিয়ার করে দেবে।

এবং কাজ শেষে আউটপুট স্ট্রিম টিকে ক্লোজ করে দিতে হবে। আউটপুট স্ট্রিম এর আরও বেশ কিছু সাব ক্লাস হলো - ক্যারেকটার স্ট্রিম

ক্যারেকটার স্ট্রিম গুলো বাইট স্ট্রিম এর মতোই কাজ করে, তবে পার্থক্য শুধু এইটুকুই যে এরা ক্যারেকটার নিয়ে কাজ করে। অর্থাৎ এগুলোকে শুধুমাত্র টেক্সট রিড এবং রাইট করার জন্যে লেখা হয়েছে। InputStream এবং OutputStream এর মতো এখানেও দুটি সুপার ক্লাস রয়েছে, যেগুলো হলো - Reader এবং Writer.

ক্যারেকটার স্ট্রিম সঠিক ভাবে বুঝতে হলে আমাদের আগে জানতে হবে ক্যারেকটার ইনকোডিং সম্পর্কে। আমরা জানি যে কম্পিউটার মূলত র (raw) জিরো-ওয়ান নিয়ে কাজ করে। কিন্তু আমরা যখন কোন টেক্সট দেখি তা কিন্তু মোটেও জিরো-ওয়ান বাইনারী ডিজিট নয়, বরং রিয়েল ক্যারেকটার গুলোই দেখি। এই জিরো-ওয়ান বাইনারী ডাটা গুলোকে ইন্টারপ্রেট করার জন্যে এক ধরনের ম্যাপিং থাকে যাকে বলা হয় ক্যারেকটার ইনকোডিং। অনেক ধরনের ক্যারেকটার ইনকোডিং থাকলেও সাধারণত ASCII ও ইউনিকোড-বেইজড ইনকোডিং গুলো নিয়ে আমাদের সমচেয়ে বেশি কাজ করতে হয়। ASCII বা আস্কি - American Standard Code for Information Interchange এর সংক্ষিপ্ত রূপ। এটি একটি ক্যারেকটার ইনকোডিং পদ্ধতি যা ইংরেজী বর্ণমালা গুলোকে নাম্বারের মাধ্যমে রিপ্রেজেন্ট করে। প্রতিটি ইংরেজী বর্ণকে একটি করে নাম্বার (০-১২৭) দেওয়া হয়। এই ইনকোডিং পদ্ধতিতে মাত্র এক বাইট এর দরকার হয়। আস্কি দিয়ে শুধুমাত্র ইংরেজী টেক্সট নিয়ে কাজ করা গেলেও পৃথিবীতে অসংখ্য ভাষা এবং বর্ণমালা রয়েছে। পৃথিবীর সব আধুনিক বর্ণমালা এবং ঐতিহাসিক দলিল গুলো নিয়ে কাজ করার জন্য একটি নতুন পদ্ধতি উদ্ভাবন করা হয়, যার নাম ইউনিকোড। এই ইউনিকোড ইমপ্লিমেন্ট করার জন্যে অনেকগুলো ক্যারেকটার ইনকোডিং স্কিম বা পদ্ধতি রয়েছে, তবে সাধারণত UTF-8, UTF-16 বেশি ব্যবহৃত হয়। UTF-8 ইনকোডিং সিস্টেম এ একটি ক্যারেকটার ১ থেকে ৪ বাইট হতে পারে এবং এটি ওয়েব পেইজ বা ইমেইল ব্যবহৃত হয়। UTF-16 এর ক্ষেত্রে তা দই বা ততোধিক বাইট হতে পারে।

অনেক সফটওয়্যার সিস্টেমই UTF ইনকোডিং স্কিম ব্যবহার করে টেক্সট স্টোর করে থাকে। যেহেতু এগুলো একটি ক্যারেকটার রিপ্রেজেন্ট করতে হলে ১ বা একাধিক বাইট দরকার হয়, সেহেতু এগুলো পড়ার সময় যদি আমরা ইনপুটস্ট্রিম ব্যবহার করে একবাইট করে পড়ি, এবং তা char এ কনভার্ট করি, তাহলে আমরা অনেক সময়ই সঠিক ভাবে ডাটা রিড করতে পারবো না। এই সমস্যা দূর করার জন্যে এবং সঠিক ভাবে টেক্সট রিড বা রাইট করার জন্যে Reader/Writer ক্লাস লেখা হয়েছে যা শুধুমাত্র টেক্সট নিয়ে কাজ করে। মনে রাখতে হবে যে, InputStream এর read() মেথড প্রত্যেকবার এক বাইট করে রিটার্ন করে আর Reader ক্লাসের read() মেথড প্রতিবার একটি করে ক্যারেকটার রিটার্ন করে। একটি বাইট এর ডায়াল ১-২৫৫ পর্যন্ত হতে পারে যেখানে একটি ক্যারেকটার এর ডায়াল ০-৬৫৫৩৫ হতে পারে। তাহলে আমরা সহজ ভাবে বলতে পারি, ইনপুট স্ট্রিম/আউটপুট স্ট্রিম র-বাইনারী ডাটা নিয়ে কাজ করে আর রিডার/রাইটার শুধুমাত্র টেক্সট নিয়ে কাজ করে।

এই পার্থক্য ছাড়া ক্যারেকটার স্ট্রিম নিয়ে কাজ করার সব স্টেপস গুলো ইনপুট/আউটপুট স্ট্রিম এর স্টেপস এর মতো।

## Read using Reader



```
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

public class ReaderExample {
    public static void main(String[] args) {
        Reader reader = null;
        try {
            reader = new FileReader("input.txt");
            int c;
            while ((c = reader.read()) != -1) {
                char ch = (char) c;
                System.out.print(ch);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            try {
                if (reader != null) {
                    reader.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

এতেও ইনপুট স্ট্রিম এর একটা করে বাইট রিড করতে হয় একটি লুপ এর মাঝে ।

## Write using Writer

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.Writer;

public class WriterExample {
    public static void main(String[] args) {
        Writer writer;
        String text = "Lorem ipsum dolor sit amet,"
            + " consectetur adipiscing elit. "
            + "Suspendisse at placerat ipsum. ";

        try {
            writer = new FileWriter("output2.txt");
            writer.write(text);
            writer.flush();
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```java

```

রাইটার এর write() মেথড দিয়ে সরাসরি স্ট্রিং রাইট করা যায়

```
**System.in, System.out, and System.error
**
```

এই তিনটি বহুল ব্যবহৃত ডাটা স্ট্রিম তবে সবচেয়ে বেশি ব্যবহৃত হয় মূলত System.out .

System.in একটি ইনপুট স্ট্রিম যা কিনা যে কোন কম্পোন প্রোগ্রামের জন্যে কিবোর্ড এর সাথে কানেক্টেড এটি System

```
```java
Scanner scanner = new Scanner(System.in);

int a = scanner.nextInt();
double d = scanner.nextDouble();

```

স্ক্যানার একটি ইউটিলিটি ক্লাস, যার সাহায্যে সহজেই আমরা কিবোর্ড থেকে ইন্টিজার বা ডাবল টাইপ ইনপুট নিতে পারি। স্ক্যানার কনস্ট্রাক্টর আর্গুমেন্ট হিসেবে একটি ইনপুট স্ট্রিম নেয়। এক্ষেত্রে আমরা System.in টি দিতে পারি যাতে করে এটি সরাসরি কিবোর্ড থেকে ডাটা পড়তে পারে।

System.out হচ্ছে System ক্লাসের একটি স্ট্যাটিক মেম্বর যা কিনা একটি প্রিন্টস্ট্রিম( PrintStream )। এটি যেকোন ডাটা কনসোল এ রাইট করে। এটিও একটি আউটপুট স্ট্রিম তবে এটি ডাটা ফরমেট করে দেখাতে সাহায্য করে। যেমন আমরা যখন কনসোল এ প্রিমিটিভ ডাটা প্রিন্ট করি, প্রিন্ট স্ট্রিম তাদের ফরমেটেড ডাটা গুলো প্রিন্ট করে, এদের বাইট ভ্যালু প্রিন্ট না করে।

`System.err` ও একটি আউটপুট স্ট্রিম যা কিনা `System.out` স্ট্রিম এর মতোই কাজ করে , তবে এটি শুধুমাত্র ইরর প্রিন্ট করার জন্যে ব্যবহার করা হয় । কিছু কিছু আইডিই এই ইরর টেক্সট গুলো লাল রং-এ প্রিন্ট করে থাকে ।

### রিডিং/রাইটিং প্রিমিটিভ ডাটা

`DataInputStream` এবং `DataOutputStream` ক্লাস দুটি প্রিমিটিভ টাইপ ডাটা কাজ করার জন্যে ব্যবহার করা হয় । এতে বেশ কিছু `readxxx()` এবং `writexxx()` মেথড রয়েছে যে গুলো ব্যবহার করে যেকোন ধরনের প্রিমিটিভ ডাটা আমরা রিড/রাইট করতে পারি ।

### উদাহরণ-

```
import java.io.DataOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class WritingPrimitivesExample {
    public static void main(String[] args) {
        String destFileName = "primitivs.data";

        try {
            DataOutputStream dos = new DataOutputStream(new FileOutputStream(destFileName));
            dos.writeInt(152);
            dos.writeDouble(4.56);
            dos.writeBoolean(true);
            dos.writeLong(Long.MAX_VALUE);

            dos.flush();
            dos.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

উদাহরণটিতে `DataOutputStream` কনস্ট্রাকট আর্গুমেন্ট হিসেবে একটি আউটপুটস্ট্রিম নেয়,এখানে যেহেতু আমরা ফাইল এ রাইট করছি,সে জন্যে `FileOutputStream` ব্যবহার করা হয়েছে ।

```
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class ReadingPrimitivesExample {

    public static void main(String[] args) {
        String sourceFile = "primitives.data";

        try {
            DataInputStream dis = new DataInputStream(new FileInputStream(sourceFile));

            int intValue = dis.readInt();
            double doubleValue = dis.readDouble();
            boolean booleanValue = dis.readBoolean();
            long longValue = dis.readLong();

            System.out.println(intValue);
            System.out.println(doubleValue);
            System.out.println(booleanValue);
            System.out.println(longValue);

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

উদাহরণটিতে `DataInputStream` কনস্ট্রাক্টর আর্গুমেন্ট হিসেবে একটি ইনপুটস্ট্রিম নেয়। যেহেতু আমরা ফাইল থেকে রিড করছি, সেহেতু ইনপুটস্ট্রিম হিসেবে `FileInputStream` ব্যবহার করা হয়েছে।

## পাঠ ১০: জাভা এন আই/ও

- পাথ
- ক্রিয়েটিং পাথ
- রিটাইডিং পাথ
- ডিরেকরি এবং ট্রি
- ফাইন্ডিং ফাইল ইন ডিরেক্টরি
- ওয়াকিং থ্রু ডিরেক্টরি
- ফাইল ক্রিয়েট এবং ডিলেট করা
- দ্রুত ফাইল রিড এবং ক্রিয়েট করা
- অ্যাসিঙ্ক্রোনাস আই/ও
- সারসংক্ষেপ

## # পাঠ ১১: জাভা কালেকশান ফ্রেমওয়ার্ক

- জাভা কালেকশান ফ্রেমওয়ার্ক ডুমিকা
- কালেকশান ইন্টারফেস
- লিস্ট
- সর্টেট লিস্ট
- ম্যাপ
- সর্টেট ম্যাপ
- নেভিগেবল ম্যাপ
- সেট
- সর্টেট সেট
- নেভিগেবল সেট
- Queue এবং Deque
- স্ট্যাক
- hashCode() এবং equals()
- সারসংক্ষেপ

## কালেকশান ফ্রেমওয়ার্ক

কালেকশান ফ্রেমওয়ার্ক জাভার টপ লেভের একটি এপিআই কনসেপ্ট । কালেকশান ফ্রেমওয়ার্ক কিছু হাইলি অপটিমাইজড ডাটা স্ট্রাকচার যার মাধ্যমে বিভিন্ন ডাটা মেমোরিতে স্টোর করতে পারি এবং প্রয়োজন মত ব্যবহার করতে পারি । মনে করুন অ্যারে নিয়ে কাজ করছেন । কোন একটি সময়ে মনে হল আপনার অ্যারের সাইজ যথেষ্ট নয় আপনার কাজের জন্য তখন কি করবেন ? মনে করুন আপনি লিংকড লিস্ট নিয়ে কাজ করছেন কোন একটা সময়ে একটা এলিমেন্ট সার্চ করার প্রয়োজন হলো , সার্চ করলেন । দেখা গেল অনেক্ষন পর রেজাল্ট জানালো যে সেই এলিমেন্ট ওই লিস্টেই নাই । এরকম নানা রকম সমস্যা এবং তার সমাধান নিয়ে যে সব ডাটা স্ট্রাকচার একত্রিত করা হয়েছে সেগুলোকেই একত্রে বলা হয় কালেকশান । কালেকশান মানে হল সমষ্টি । এটি এমন কিছু ডাটা স্ট্রাকচারের সমষ্টি যেগুলার প্রতিটিই বিভিন্ন ডাটাকে সমষ্টিত করে রাখে । হ্যা অ্যারেকেও লো লেভেল এক প্রকার কালেকশান বলা যেতে পারে তবে মডার্ন কালেকশান ফ্রেমওয়ার্কের মাঝে এটিকে ধরা হয়না ।

## কালেকশান ইন্টারফেস (Collection Interface)

ইন্টারফেস কি সেটি আপনারা খুব ভালোভাবেই জানেন । যদি না যেতে থাকেন তবে চ্যাপটার ৫.১ পড়ে আসুন । কালেকশান একটি ইন্টারফেস । যেই ইন্টারফেসের মধ্য বলে দেওয়া হয়েছে কোন একটি ক্লাসকে কালেকশান ফ্রেমওয়ার্কের অন্তর্গত হতে গেলে কি কি বৈশিষ্ট্য থাকতেই হবে । কালেশনগুলো সাধারণত `java.util` প্যাকেজের অন্তর্গত ।

সবার আগে আমাদের জানা প্রয়োজন কেনই বা আমরা কালেকশানস নিয়ে কাজ করবো ? এটি না নিয়েও তো কাজ করা যেতো । তাহলে কালেকশান কেন !

ওয়েল , আপনাদের কিছুটা উত্তর আমি আগেই দিয়ে দিয়েছি ।

১) অ্যারে নিয়ে কাজ করার সময় আপনি ফিক্সড লেন্থের বাইরে কাজ করতে পারতেন না । অ্যারের বাউন্ডারি ফিক্সড এবং এটি বাড়ানো বা কমানোর কোন সুযোগ নেই রানটাইমে । কালেকশান এই সমস্যার সমাধান করেছে । এটির সাইজ আপনার প্রয়োজন মত বাড়াতে এবং কমাতে পারবেন ।

২)লিংক লিস্ট নিয়ে কাজ করার সময় আপনি ইনডেক্সের সুবিধা পাবেন না । এটা একটা বড় সমস্যার কারন, কালেকশনে আপনি ইন্ডেক্স সুবিধা পাবেন ।

৩)কেবল প্রিমিটিভ নয়, সকল প্রকার অবজেক্ট এমনি একটি কালেকশনের মাঝে আরেকটি কালেকশন নিয়ে কাজ করার মত ফ্লেক্সিবিলিটি পাবেন ।

এছাড়া আরো বহুত সুবিধা আছে যেগুলো কাজ করতে করতে বুঝে যাবেন ।

নিচে কালেকশান ফ্যামিলি টি দেখানোর চেষ্টা করা হল ।

```
Collection<Interface>
    Set<Interface>
        HashSet
        LinkedHashSet
        SortedSet<Interface>
        TreeSet
    List<Interface>
        ArrayList
        Vector
        LinkedList
    Queue<Interface>
        LinkedList
        PriorityQueue

Object
    Arrays
    Collections

Map<Interface>
    Hashtable
    LinkedHashMap
    HashMap
    SortedMap<Interface>
    TreeMap
```

## লিস্ট ( List )

লিস্ট `List` একটি ইন্টারফেস যেটি সরাসরি `Collection` ইন্টারফেসকে এক্সটেন্ড করেছে । এটি যেহেতু একটি ইন্টারফেস ( `interface` ) তাই আমরা সরাসরি এটার কোন অবজেক্ট বা ইন্সট্যান্স ক্রিয়েট করতে পারবো না । এজন্য অবশ্য আমাদের চিন্তার খুব বেশি কারন নেই । `List` ইন্টারফেসকে ইমপ্লিমেন্ট করেছে `ArrayList` , `Vector` এবং `LinkedList` ক্লাস । আমরা খুব সহজে এগুলার মাধ্যমে `List` এর অবজেক্ট তৈরি করতে পারি । `List` হল আনসর্টেড অবজেক্ট কনটেইনার যেটি ডাটা ডুপ্লিকেসি সাপোর্ট করে । মানে একই ডাটা একাধিকবার থাকতে পারে লিস্টের মাঝে ।

`List` ডিক্লেয়ার করার নানা ধাপঃ

ধাপ ১ঃ

```
import java.util.ArrayList;
import java.util.List;
import java.util.LinkedList;
import java.util.Vector;
public class Main {
    public static void main(String[] args) {
        List list, list2, list3;
        list = new ArrayList();
        list2 = new LinkedList();
        list3 = new Vector();
    }
}
```

## ধাপ ২৪

```
import java.util.ArrayList;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List list1 = new ArrayList();
        ArrayList list2 = new ArrayList();
    }
}
```

এমনকি কেউ চাইলে কালেকশনের অবজেক্ট নিয়েও কাজ করতে পারেন সেক্ষেত্রে যেটি করতে হবে ।

```
import java.util.ArrayList;
import java.util.Collection;
public class Main {
    public static void main(String[] args) {
        Collection c = new ArrayList();
    }
}
```

ওকে অনেক হয়েছে । এবার কাজের কথায় আসা যাক । `List` নিয়ে কিভাবে কাজ করা যায় সেটাইতো জানা হলোনা এখনো ! ওকে আর বেশি বক বক করে আপনাদের ধৈর্য্যের পরীক্ষা নিবনা । প্রথমে আমরা দেখবো কিভাবে একটি লিস্টে ডাটা অ্যাড বা অ্যাসাইন করতে হয় ।

## লিস্টে ডাটা ইনসার্ট(Insert into List)



```
import java.util.ArrayList;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        list.add(7);
        list.add(5);
        list.add(13);
        list.add(17);
        list.add(3);
    }
}
```

উপরোক্ত কোডে `<Integer>` দিয়ে বোঝানো হয়েছে এই লিস্টটি কেবল ইন্টিজার টাইপ ডাটার জন্য কাজ করবে। এটিকে জেনেরিক বলা হয়। চ্যাপটার ৮ এ আপনাদের এ বিষয়ে জানা কথা। তারপর `ArrayList` এর কনস্ট্রাক্টর দিয়ে `list` অবজেক্টকে ইন্সট্যানশিয়েট করা হয়েছে। `add` মেথড এই লিস্টে একটি একটি করে ডাটা অ্যাড করে এবং একটি বুলিয়ান ভ্যালু রিটার্ন করে। যদি কোন কারনে কোন ডাটা অ্যাড করতে ব্যর্থ হয় তবে `false` ভ্যালু রিটার্ন করে।

`add(int index, E element)` মেথডটি যেকোন একটি এলিমেন্ট লিস্টের নির্দিষ্ট ইনডেক্সে ইনসার্ট করে। `addAll(Collection<? Extends E> c)` মেথডটি ইনপুট প্যারামিটার হিসাবে অন্য কোন একটি লিস্ট বা কালেকশন নিয়ে তার প্রতিটি এলিমেন্ট একটু একটি করে এই লিস্টে ইনসার্ট করে দেয়। `addAll(int index, Collection<? Extends E> c)` মেথডটি ঠিক আগের মতই কাজ করে। নির্দিষ্ট ইনডেক্স থেকে অন্য একটি কালেকশনকে ইনজেক্ট করতে থাকে নতুন লিস্টের মাঝে।

### লিস্ট থেকে ডাটা রিড করা(Read from List)

উপরের কোড সেগমেন্টটি মনে করলাম আছে। আমরা কেবল ডাটা রিড করার জন্য কোডটি লিখবো। আমরা বেশ কয়েকভাবে দেখবো যে কিভাবে একটি লিস্ট থেকে ডাটা রিড করা যায় এবং এর মাধ্যমে আরো কিছু মেথড সম্পর্কে জেনে নিব।

পদ্ধতি ১ঃ

```
for(int i=0; i<list.size(); i++){
    System.out.println(list.get(i));
}
```

এটি একেবারে চীরাচরিত পদ্ধতি হলেও বেশ কার্যকর। এখানে অ্যারের মত ইন্ডেক্স নিয়ে খেলা করার সুযোগ আছে। `size` মেথডটি একটি ইন্টিজার নাম্বার রিটার্ন করে যেটি নির্দেশ করে এই লিস্টে মোট কতগুলো এলিমেন্ট আছে। মানে এই লিস্টের সাইজ কত। `get` মেথডটি একটি ইন্টিজার নাম্বার ইনপুট হিসাবে নেয় এবং সেই ইন্ডেক্সের এলিমেন্টটি রিটার্ন করে। এমন কোন ইনডেক্স যদি ইনপুট হিসাবে দেওয়া হয় যেটি এই লিস্টের সাইজের মাঝে পড়েনা তাহলে `java.lang.IndexOutOfBoundsException` এক্সেপশন থ্রো করবে।

পদ্ধতি ২ঃ

```
for(int x: list){

    System.out.println(x);

}
```

এটি একেবারে চীরাচরিত পদ্ধতি থেকে একটু আধুনিক । এনহ্যান্স ফর লুপ । এই পদ্ধতিতে এনহ্যান্স ফর লুপ লিস্ট থেকে একটি একটি এলিমেন্ট প্রতি এলিমেন্টে পিক করে `x` ভ্যারিয়েবলের মাঝে রেফার করছে এবং সেটিই আমাদের সামনে প্রদর্শিত হচ্ছে ।

পদ্ধতি ৩ঃ

```
Iterator it = list.iterator();
while(it.hasNext()){
    System.out.println(it.next());
}
```

এটি আরো একটু মডার্ন পদ্ধতি । এই পদ্ধতিতে প্রথমেই `Iterator` ইন্টারফেসের একটি ইন্সট্যান্স ক্রিয়েট করা হচ্ছে লিস্টের `iterator` মেথডটি কল করে । এই মেথডটি একটি ইটারেটর অবজেক্ট রিটার্ন করে যেটি এই লিস্টে টপ টু বটম ইটারেট করতে পারে । ইটারেটরের `hasNext` মেথডটি একটি বুলিয়ান ভ্যালু রিটার্ন করে । প্রতি ইটারেশনের পর সে বলে দেয় এর পর আর কোন এলিমেন্ট অবশিষ্ট আছে কি না । ইটারেটরের `next` মেথডটি প্রতিবার নতুন একটি এলিমেন্ট প্রভাইড করে এবং এটির কাউন্টার পয়েন্টার তার পরের ইনডেক্সে শিফট করে । যাতে করে পরেরবার নতুন একটি এলিমেন্ট রিটার্ন করতে পারে ।

পদ্ধতি ৪ঃ

```
list.forEach((x) -> {
    System.out.println(x);
});
```

হাল আমলের আলোচিত পদ্ধতি । এটিকে বলা হয় ফাংশনাল অপারেশন । জেডিকে ৮ এ এটিকে পরিচিত করানো হয়েছে । অনেকটা ফাংশনাল প্রোগ্রামিং এর মত করেই ডিজাইন করা করা । `forEach` মেথডটি একটি একটি করে এলিমেন্ট ট্রান্সার্স করে যায় এবং তাকে যে কাজ করতে বলা হয় ঠিক সেই কাজটিই করে বসে থাকে । :D দারুন মজার এই ফাংশনাল অপারেশন ।

লিস্টের ভ্যালু রিপ্লেস করা

কোন একটি লিস্ট থেকে খুব সহজেই একটি ভ্যালু রিপ্লেস করে দেওয়া যায় । `set(int index, E element)` মেথডটি ২ টি ইনপুট প্যারামিটার নেয় । প্রথমে যে ইনডেক্সের ভ্যালু রিপ্লেস করতে হবে সেটি এবং তার পরে যে অবজেক্ট দিয়ে সেই স্থান পূরন করতে হবে সেটি ।

```
list.set(2, Integer.MAX_VALUE);
```

লিস্ট থেকে ডিলিট করা

খুব প্রচলিত ২ উপায়ে লিস্ট থেকে কোন একটি এলিমেন্ট ডিলিট বা রিমুভ করে দেওয়া যায়। একটি হল কোন একদি এলিমেন্ট বা অবজেক্ট কোন ইনডেক্সে আছে সেটা জানা এবং সেই ইনডেক্সকে রিমুভ করে দেওয়া। অথবা যে অবজেক্টটি রিমুভ করতে চাওয়া হচ্ছে সেই অবজেক্টটি দিয়ে বলা সেটি ডিলিট করতে। চলুন দেখি সেটি কিভাবে করা যায়ঃ

পদ্ধতি ১ঃ

```
list.remove(2);
```

এই পদ্ধতি আপনি প্রিমিটিভ ইন্টিজার নাম্বার নাম্বার ইনপুট প্যারামিটার হিসাবে পাস করছেন। অর্থাৎ `remove` মেথডটি এটিকে ইনডেক্স হিসাবে বিবেচনা করবে। যদি 2 নাম্বার ইনডেক্সে অন্য কোন নাম্বার থাকে এবং ২ লিস্টে উপস্থিত থাকে তার পরেও সে ২ নাম্বার ইনডেক্সের ড্যালুটিকে রিমুভ করবে এবং ওই ইনডেক্সের ড্যালুটি রিটার্ন করবে।

পদ্ধতি ২ঃ

```
list.remove(new Integer(13));
```

এই পদ্ধতিতে আপনার লিস্টটি যে টাইপের অবজেক্ট কনটেইন করছে সেই টাইপের একটি অবজেক্ট দিলে সেটিকে ডিলিট করার চেষ্টা করবে। যদি উক্ত অবজেক্ট উপস্থিত থাকে ডিলিট করবে এবং `true` ড্যালু রিটার্ন করবে অন্যথায় `false` রিটার্ন করবে।

লিস্ট সম্পর্কিত কিছু মেথড(**Some methods of List**)

`clear` মেথডটি উক্ত লিস্ট থেকে সব এলিমেন্ট রিমুভ করে দেয়। `contains` মেথডটি একটি অবজেক্ট ইনপুট হিসাবে নেয় এবং চেক করে যে উক্ত অবজেক্টের লিস্টে প্রেজেন্ট কি না। `indexOf` মেথডটি একটি অবজেক্ট ইনপুট হিসাবে নেয় এবং যদি সেই অবজেক্টটি ওই লিস্টে প্রেজেন্ট থাকে তবে তার ইনডেক্স রিটার্ন করে। অন্যথায় -১ রিটার্ন করে। `sort` নামক একটি মেথড আছে যেটি ইনপুট প্যারামিটার হিসাবে কম্পারেটর অবজেক্ট নিয়ে লিস্টটি সেই অনুযায়ী সর্ট করে।

এরকম আরো বেশ কিছু মেথড এবং তাদের বিস্তার ব্যাখ্যা ওরাকলের অফিসিয়াল ডকুমেন্টেশন সাইটে পাওয়া যাবে। আগ্রহীরা সেখান থেকে দেখে নিতে পারেন। লিংকঃ

<http://docs.oracle.com/javase/8/docs/api/java/util/List.html>

লিস্ট সর্ট করা(**Sort a List**)

কোন একটি লিস্টকে সর্ট করার চেয়ে সহজ বিষয় আর কিছু হতেই পারেনা। তবে সমস্যা হল একটি লিস্টকে সর্ট করার নানাবিধ উপায় থাকায় আপনি কনফিউজ হয়ে যেতে পারেন যে আসলে কক্ষন কোন পদ্ধতিতে সর্ট করবেন। আমি নিজেও মাঝে মাঝে কনফিউজ হয়ে যাই। যাইহোক আমরা লিস্ট সর্টিং এর একেবারে বেসিক থেকে ধীরে ধীরে সামনের দিকে এগিয়ে যাব। বলে রাখা ভালো আমরা এখানে বেসিক্যালি ২ প্রকারের সর্টিং টেকনিক দেখবো এবং তাদের আবার ২ প্রকার সাব সর্টিং টেকনিক দেখবো। আগ্রহীরা আরো কিছুটা ঘাটাঘাটি করলে আরো অনেক কিছুই জানতে পারবে।

পদ্ধতি ১ঃ

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class Main {

    public static void main(String[] args) {

        List<Integer> list = new ArrayList<>();
        list.add(7);
        list.add(5);
        list.add(13);
        list.add(17);
        list.add(3);

        Collections.sort(list);

        list.forEach((x) -> {
            System.out.println(x);
        });
    }
}
```

এখানে আমরা `Collections` ক্লাসের একটি মেথড `sort` যেটি ইনপুট প্যারামিটার হিসাবে একটি লিস্ট অবজেক্ট নেয় এবং সেটিকে ইনপ্লেস সর্ট করে দেয়। অর্থাৎ এই মেথডের রিটার্ন টাইপ ডয়েড। এবং এটি অ্যাসেন্ডিং (ছোট থেকে ক্রমান্বয়ে বড়) অর্ডারে সর্ট করে।

আমরা যদি ডিসেন্ডিং অর্ডারে সর্ট করতে চাই তবে আমাদের আরেকটু কাজ বেশি করতে হবে। আর সেটি হল `sort` নামক মেথডে আরেকটি প্যারামিটার পাস করতে হবে যেটি আসলে একটি `Comparator` অবজেক্ট। যেটার মাধ্যমে আমরা বলে দিব যে আসলে সর্টটি কোন অর্ডারে হবে বা কোন এলিমেন্টের সাপেক্ষে হবে। কোডটিকে সেক্ষেত্রে আমরা এভাবে লিখতে পারতাম,

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class Main {

    public static void main(String[] args) {

        List<Integer> list = new ArrayList<>();
        list.add(7);
        list.add(5);
        list.add(13);
        list.add(17);
        list.add(3);

        Comparator<Integer> comparator = new Comparator<Integer>() {
            @Override
            public int compare(Integer t1, Integer t2) {
                return t2-t1;
            }
        };

        Collections.sort(list, comparator);

        list.forEach((x) -> {
            System.out.println(x);
        });
    }
}
```

উল্লেখ্য এখানে `Comparator` একটি ইন্টারফেস এবং `compare` একটি `abstract` মেথড তাই আমাদের এটিকে ইমপ্লিমেন্ট করতে হয়েছে। `compare` মেথডটি একটি ইন্টিজার নাম্বার রিটার্ন করে। দুটি অ্যাক্টিবিউটের মাঝে কম্পেয়ার করে পজেটিভ, নেগেটিভ বা শূন্য রিটার্ন করে। শূন্য রিটার্ন করা মানে দুটি সমান। পজেটিভ রিটার্ন করা মানে প্রথমটি বড় আর অন্যথায় ছোট। আমরা আলাদা ভাবে `Comparator` এর অবজেক্ট ক্রিয়েট না করেও কাজটি করতে পারতাম ইনপ্লেসে। সেক্ষেত্রে এরকম হতে পারত,

```
Collections.sort(list, new Comparator<Integer>() {
    @Override
    public int compare(Integer t, Integer t1) {
        return t1-t;
    }
});
```

বর্তমান সময় যেহেতু ল্যাম্বডা এক্সপ্রেসনের যুগ চলছে তাই এটিকে আরো সহজে এবং খুব সংক্ষেপে এভাবেও লেখা যেত,

```
Collections.sort(list, (Integer t1, Integer t2) -> t2-t1);
```

কাজ বেসিক্যালি সব একই ভাবে করছে ।

এতক্ষন আমরা দেখলাম প্রিমিটিভ টাইপের ডাটার একটি লিস্ট সর্ট করা । এমনতো হতেই পারে যে আপনার কাছে একটি কাষ্ট টাইপের অবজেক্ট কনটেইন করে এমন একটি লিস্ট সর্ট করতে হবে ওই অবজেক্টের নির্দিষ্ট কোন এক বা একাধিক প্রপার্টির সাপেক্ষে । সেক্ষেত্রে করণীয় কি সেটা এবার চলুন দেখে ফেলি ।

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Employee {

    int id;
    String name;
    int age;
    int salary;

    public Employee(int id, String name, int age, int salary) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
}

public class Main {

    public static void main(String[] args) {

        List<Employee> list = new ArrayList<>();
        list.add(new Employee(1, "Abul", 27, 35000));
        list.add(new Employee(2, "Babul", 25, 37000));
        list.add(new Employee(3, "Kabul", 29, 30000));
        list.add(new Employee(4, "Mofiz", 24, 36000));
        list.add(new Employee(5, "Hafiz", 28, 34000));

        Collections.sort(list, new Comparator<Employee>(){
            @Override
            public int compare(Employee t, Employee t1) {
                return t.age - t1.age;
            }
        });

        list.forEach((x) -> {
            System.out.println(x.salary);
        });
    }
}

```

ঠিক আগের মতই কাজ করতে পারবেন। তবে এখানে অবজেক্ট নিয়ে কাজ করতে হবে। এবং অবজেক্টের কোন ফিল্ডের রেসপেক্টে সর্ট করতে চাচ্ছেন সেটাও ডিফাইন করে দিতে হবে। অ্যাসেন্ডিং বা ডিসেন্ডিং যেকোন ভাবেই সর্ট করতে পারবে। এমনকি একাধিক ফিল্ডের রেসপেক্টে যদি সর্ট করতে চান সেটাও করতে পারবে। আপনারা চাইলে ল্যাম্বডা এক্সপ্রেশন ব্যবহার করতে পারতেন সেক্ষেত্রে এরকম হতো কোডটি,

```
Collections.sort(list, (Employee t, Employee t1) -> t1.age - t.age); //descending
```

এবার আমরা একটু ভিন্ন একটা পদ্ধতি দেখবো। যদি এমন হয় যে আমাদের এই ক্লাসটি প্রায়ই সর্ট করতে হয় এবং নির্দিষ্ট একটা অর্ডারে সর্ট করতে হয়, সেক্ষেত্রে আমরা একটা বিশেষ কাজ করতে পারি। আমরা `Comparable` ইন্টারফেসটি ইমপ্লিমেন্ট করতে পারি। `Comparable` ইন্টারফেসের মাঝে `compareTo` মেথডটি অডাররাইড করলেই কাজ শেষ। অ্যাসেন্ডিং বা ডিসেন্ডিং আগের মতই। তবে চলুন দেখি কিভাবে সেটি করা যায় সেটি দেখে ফেলি।



```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Employee implements Comparable<Employee> {

    int id;
    String name;
    int age;
    int salary;

    public Employee(int id, String name, int age, int salary) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    @Override
    public int compareTo(Employee t) {
        return this.age - t.age;
    }
}

public class Main {

    public static void main(String[] args) {

        List<Employee> list = new ArrayList<>();
        list.add(new Employee(1, "Abul", 27, 35000));
        list.add(new Employee(2, "Babul", 25, 37000));
        list.add(new Employee(3, "Kabul", 29, 30000));
        list.add(new Employee(4, "Mofiz", 24, 36000));
        list.add(new Employee(5, "Hafiz", 28, 34000));

        Collections.sort(list);

        list.forEach((x) -> {
            System.out.println(x.id + ", " + x.name + ", " + x.age + ", " + x.salary);
        });
    }
}

```

এভাবে খুব সহজেই আমরা একটি অবজেক্টের লিস্ট সর্ট করতে পারি। তবে একটি বিষয় লক্ষ করার মত বিষয় হচ্ছে যে আপনাদের যে ২ প্রকারের সর্ট দেখানো হয়েছে অবজেক্টের লিস্টের ক্ষেত্রে এই দুই প্রকার কিন্তু একত্রেও ব্যবহার করতে পারবেন। তবে সেক্ষেত্রে প্রায়োরিটি পাবে ক্রোজার ফাংশন। চলুন দেখি বিষয়টা কি একটু দেখে নেই।

```

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class Employee implements Comparable<Employee> {

    int id;
    String name;
    int age;
    int salary;

    public Employee(int id, String name, int age, int salary) {
        this.id = id;
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    @Override
    public int compareTo(Employee t) {
        return this.age - t.age;
    }
}

public class Main {

    public static void main(String[] args) {

        List<Employee> list = new ArrayList<>();
        list.add(new Employee(1, "Abul", 27, 35000));
        list.add(new Employee(2, "Babul", 25, 37000));
        list.add(new Employee(3, "Kabul", 29, 30000));
        list.add(new Employee(4, "Mofiz", 24, 36000));
        list.add(new Employee(5, "Hafiz", 28, 34000));

        Collections.sort(list, ((Employee e1, Employee e2) -> (e2.salary + e1.salary)));

        list.forEach((x) -> {
            System.out.println(x.id + ", " + x.name + ", " + x.age + ", " + x.salary);
        });
    }
}

```

এখানে যদিও `Employee` ক্লাস `Comparable` ইন্টারফেস ইমপ্লিমেন্ট করেছে এবং `compareTo` মেথডে বলে দেওয়া হয়েছে `age` এর অ্যাসেন্ডিং অর্ডারে সর্ট করতে হবে কিন্তু এটি সর্ট করবে `salary` এর ডিসেন্ডিং অর্ডারে। কেন সেটা করছে সেটা নিশ্চয় বুঝতে পেরেছেন।

আপনারা চাইলে কিন্তু `List` এর অন্তর্গত `sort` মেথড ব্যবহার করেও সর্ট করতে পারতেন। সেটা করার জন্য খুব বেশি কিছুই করতে হতনা। সেটি অলরেডি আপনারা জানেন কিভাবে সেটি করা যায়।

```
list.sort(new Comparator<Employee>(){
    @Override
    public int compare(Employee t, Employee t1) {
        return t1.age - t.age;
    }
});
```

বেসিক্যালি এভাবে খুব সহজেই একটি লিস্ট সর্ট করা যায় । আপনারা আরো বেশি আগ্রহী হলে ওরাকলের ডকুমেন্টেশন পড়তে পারেন । আরো বেশি পরিষ্কার হবে ধারণা ।

-----চলবে-----

## পাঠ ১২: জাভা জেডিবিসি

- জেডিবিসি ভূমিকা
- জেডিবিস ডাইভার এবং টাইপস
- কানেকশান
- কুয়েরি
- রেজাল্টসেট
- প্রিপেয়ার স্ট্যাটমেন্ট
- ট্রানসেকশান
- সারসংক্ষেপ

## পাঠ ১৩: জাভা লগিং

- সাধারণ ব্যবহার
- লগার
- লাগার হাইআরকি
- লগ লেভেলস
- ফরমেটারস
- ফিল্টারস
- কনফিগারেশন
- সারসংক্ষেপ

## পাঠ ১৪: ডিবাগিং

- ডিবাগিং ফ্লো
- ডিবাগার দিয়ে ডিবাগিং
- ব্রেক পয়েন্ট এবং ভ্যারিয়েবলস
- স্কোপস এবং স্টেপস
- ডিবাগিং টিপস
- সারসংক্ষেপ

## পাঠ ১৫: গ্রাফিক্যাল ইউজার ইন্টারফেইস

- সুইং
- কনটেইনার, কম্পোনেন্ট, ইভেন্ট, লিসেনার এবং লেআউট
- কোডিং উদাহরণ এবং ব্যাখ্যা
- সারসংক্ষেপ

## পাঠ-১৬: থ্রেড

- থ্রেড কি
- থ্রেড কনস্ট্রাকশন
- রানেবল ইন্টারফেস
- থ্রেড মেথড
- থ্রেড ইন্টারপশান
- থ্রেড স্টপ
- থ্রেড স্কেজিওলিং
- থ্রেড সেইফটি
- থ্রেড পুল
- সারসংক্ষেপ



## পাঠ ১৭: নেটওয়ার্কিং

- সকেট
- ক্লায়েন্ট/সার্ভার
- টিসিপি
- ইউডিপি
- পোর্ট
- ইউআরএল
- একটি চ্যাট প্রোগ্রাম কোডিং উদাহরণ
- সারসংক্ষেপ

## পাঠ ১৮: জাভা কনকারেন্সি

- ভূমিকা
- বেনিফিট
- কন্সট
- রেস কন্ডিশান এবং ক্রিটিকাল সেকশান
- থ্রেড সেইফটি এবং শেয়ার্ড রিসোর্স
- থ্রেড সেইফটি এবং ইমুটাবিলিটি
- সিনক্রোনাইজেশান ব্লক
- ডেড লক এবং প্রডেনশান
- রিড/রাইট লকস
- সেমাফোর
- ব্লকিং কিও
- থ্রেড পোল
- কন্টোলিং এক্সিকিউশান
- মডেলিং টাস্ক
- ScheduledThreadPoolExecutor
- ফর্ক/জয়েন ফ্রেমওয়ার্ক
- একটি সিম্পল ফর্ক/জয়েন উদাহরণ
- ফর্কজয়েনটাস্ক এবং ওয়ার্ক স্টিলিং
- Parallelizing problems
- জাভা মেমরী মডেল
- সারসংক্ষেপ

## পাঠ ১৯: ক্লাস ফাইল এবং বাইটকোড

- ক্লাসলোডিং এবং ক্লাস অবজেক্ট
- MethodHandle
- MethodType
- Looking up method handles
- Examining class files
- বাইটকোড
- disassembling a class
- রানটাইম ইনভারনমেন্ট
- অপকোড
- লোড অপকোড
- অ্যারিথম্যাটিক অপকোড
- Execution control opcodes
- Invocation opcodes
- Platform operation opcodes
- উদাহরণ—string concatenation
- Invokedynamic
- ইনভোটাইনামিক কি এবং কিভাবে কাজ করে
- উদাহরণ

## পাঠ ২০: Understanding performance tuning

- টারমিনলজি
- Latency
- Throughput
- Utilization
- Efficiency
- Capacity
- Scalability
- Degradation
- প্রাগমেটিং এপ্রোচ
- মুরস-ল
- মেমরী ল্যাটেন্সি হাইআরকি
- কেন জাভা পারফরমেন্স টিউনিং কেন কঠিন
- হার্ডওয়্যার ক্লকস
- কেস স্টাডি
- গারবেজ কালেক্টর
- Mark and sweep
- Jmap
- JVM প্যারামিটার
- রিডিং জিসি লগস
- ডিজুয়ালভিএম
- এসকেপ এনালাইসিস
- Concurrent Mark-Sweep
- G1—Java's new collector
- হটস্পট দিয়ে জিট(JIT) কম্পাইলেশন
- Inlining methods
- Dynamic compilation and monomorphic calls
- Reading the compilation logs

## পাঠ ২১: মডার্ন জাভা ইউজেস

- রপিড ওয়েব ডেভেলপমেন্ট
- জাভা ফ্রেমওয়ার্ক
- স্প্রিং
- GWT
- Struts 2
- Wicket
- Tapestry
- JSF
- Vaadin
- Play
- Plain old JSP /Servlet
- অন্যান্য জেভিএম ল্যাংগুয়েজ
- গ্রাইলস
- ক্রোজার
- স্কেলা
- আরলেং
- সারসংক্ষেপ

## অনুশীলন # ১

Design a class named Point which will model a 2D point with `x` and `y` coordinates.

1. Two instance variable variables `x (int)` and `y (int)`
2. A "no-argument" (or "no-arg") constructor that construct a point at `(0, 0)`.
3. A constructor that constructs a point with the given `x` and `y` coordinates.
4. Getter and setter for the instance variables `x` and `y`.
5. A method `setXY()` to set both `x` and `y`.
6. A `toString()` method that returns a string description of the instance in the format `"(x, y)"`.
7. A method called `distance(int x, int y)` that returns the distance from this point to another point at the given `(x, y)` coordinates.
8. An overloaded `distance(Point another)` that returns the distance from this point to the given `Point` instance another.

For example -

```
package bd.com.howtocode.java;

/**
 * @author Bazlur Rahman Rokon
 * @date 6/19/15.
 */

public class Point {
    private int x;
    private int y;

    public Point() {
        x = 0;
        y = 0;
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }
}
```

```
public int getY() {
    return y;
}

public void setY(int y) {
    this.y = y;
}

public double distance(int x, int y) {
    int xDiff = this.x - x;
    int yDiff = this.y - y;
    return Math.sqrt(xDiff * xDiff + yDiff * yDiff);
}

public double distance(Point p2) {
    return distance(p2.getX(), p2.getY());
}

@Override
public String toString() {
    return "(" + x + ", " + y + ")";
}
}
```

Now write a program that allocates 10 points in an array of `Point`, and initializes to (1, 1), (2, 2), ... (10, 10).