Programming Fundamentals

Agenda

Code Optimization

Coding Best Practices

Self Healing

Memory Management

Checkstyle

Code Optimization

Code Optimization

- Code optimization is an approach to enhance
    - Performance
    - Quality
    - Efficiency
    - Consumes less memory
    - Execution speed
    - Reducing the size of the application (LOC)
    - Perform fewer input / output operations
    - Reduces CPU time and network bandwidth

Code Optimization contd…

Performance Tuning

- ❖ Performance tuning is the improvement of system performance.
- ❖ Performance optimization, also known as "performance tuning", is an iterative approach to making and then monitoring modifications to an application and its database.
- ❖ The motivation of such activity is called a performance problem, which can be either real or anticipated.
- ❖ Most systems will respond to increased load with some degree of decreasing performance.

Code Optimization contd...

System Limitations and What to Tune?

Three resources that limit all applications

- CPU speed and availability
- System Memory
- Disk input/output

When tuning an application, the first step is to determine which of these is causing the application to run too slowly.

Code Optimization contd...

Optimizations are classified into high-level and low-level optimizations.

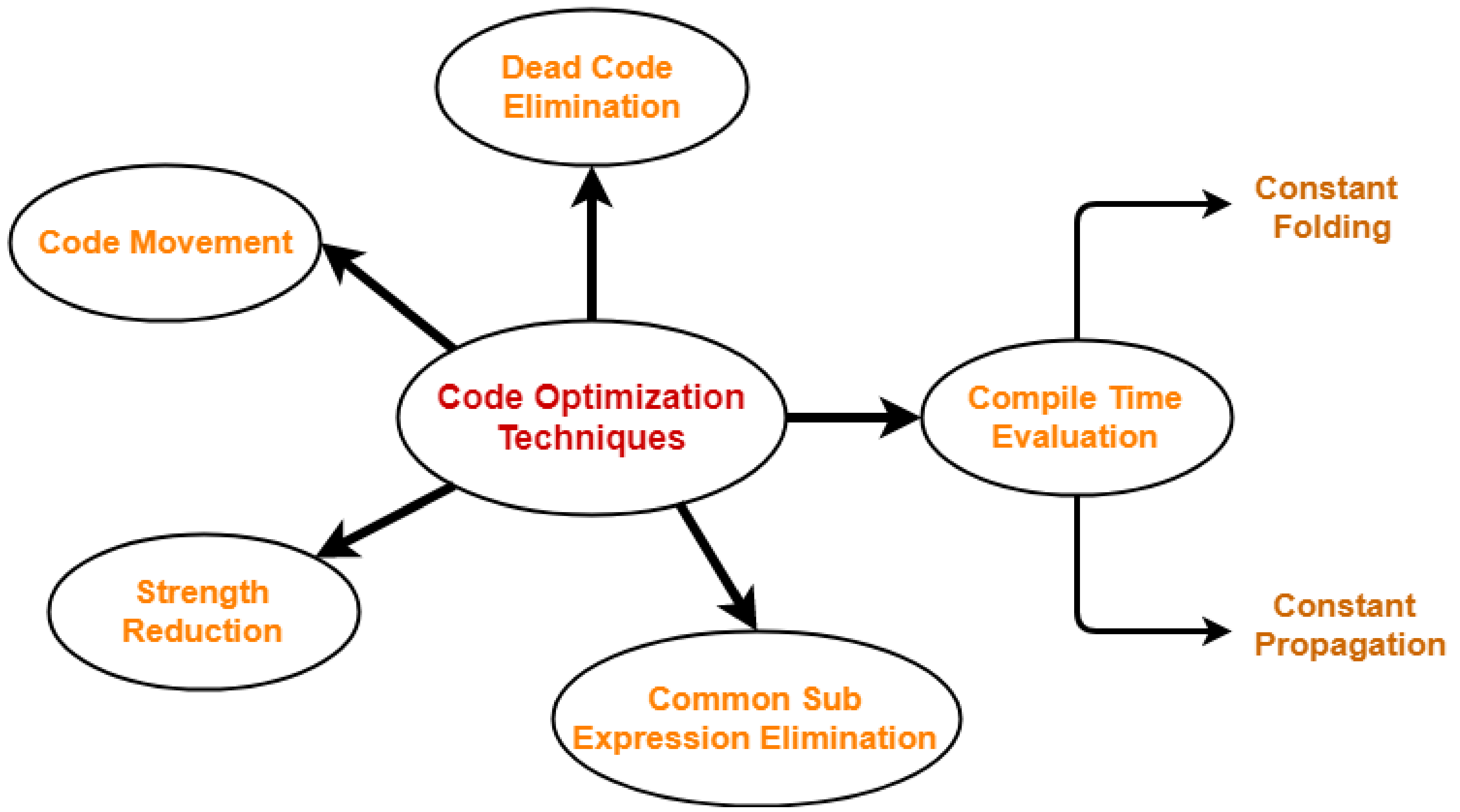
High-level optimizations :

It is usually performed by the programmer, who handles abstract entities (functions, procedures, classes, etc.) and keeps in mind the general framework of the task to optimize the design of a system

Low-level optimizations:

It is usually performed at the stage when source code is compiled into a set of machine instructions, and it is at this stage that automated optimization is usually employed

Code Optimization Techniques contd...



Code Optimization Techniques contd…

1. Compile Time Evaluation

Two techniques that are under compile time evaluation is,

- Constant Folding
- Constant Propagation

Code Optimization Techniques contd…

1.A Constant Folding:
It is a technique of evaluating the expressions operand which are know to be constant at the compile time itself

Those expressions are then replaced with their respective results.

Example: length=22/7

This technique evaluates the expression 22/7 at compile time. The expression is then replaced with its result 3.14 which saves the time at run time.

Code Optimization Techniques contd…

1.B Constant Propagation

- In this technique, if some variable has been assigned with some constant value, then it replaces that variable with its constant value in the further program during compilation.

- The value of variable must not get alter in between.

Example

Area of Circle = pi * r * r
pi = 3.14, r = 5
Then,
A = 3.14 * 5 * 5

•This technique substitutes the value of variables 'pi' and 'radius' at compile time.
•It then evaluates the expression 3.14 x 10 x 10.
•The expression is then replaced with its result 314.
•This saves the time at run time.

Code Optimization Techniques contd...

2. Common Sub-Expression Elimination

- The expression that has been already computed before and appears again in the code for computation is called as Common Sub-Expression.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

Code Before Optimization	Code After Optimization

S1 = 4 x i
S2 = a[S1]
S3 = 4 x j
S4 = 4 x i **// Redundant Expression**
S5 = n
S6 = b[S4] + S5
	S1 = 4 x i
	S2 = a[S1]
	S3 = 4 x j
	S5 = n
	S6 = b[S1] + S5

Code Optimization Techniques contd…

3. Code Movement

- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

Code Before Optimization	Code After Optimization

for (int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	a[j] = 6 x j;
	}

Code Optimization Techniques contd...

Loop Optimization

❖ Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops.

❖ It plays an important role in improving cache performance and making effective use of parallel processing capabilities.

Frequency Reduction (Code Motion):

❖ In frequency reduction, the amount of code in loop is decreased.

❖ A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.

Code Optimization Techniques contd…

Dead Code Elimination

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

Code Before Optimization	Code After Optimization

i = 0 ;
if (i == 1)
{
a = x + 5 ;
}
	i = 0 ;

Code Optimization Techniques contd...

Strength Reduction

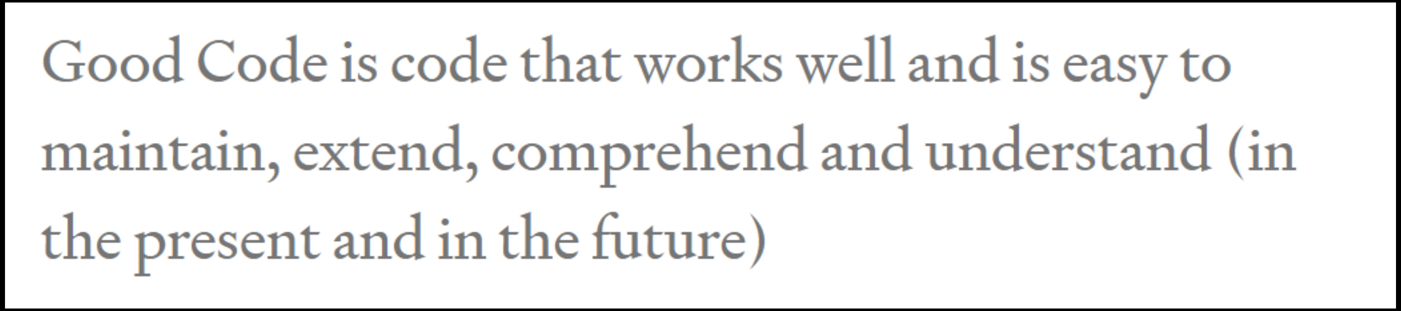
- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

Code Before Optimization	Code After Optimization
B = A x 2	B = A + A

- The expression "A x 2" is replaced with the expression "A + A".
- This is because the cost of multiplication operator is higher than that of addition operator.

Coding Best Practices

What is a good code ?



Good Code is code that works well and is easy to maintain, extend, comprehend and understand (in the present and in the future)

Characteristics of a Good Code

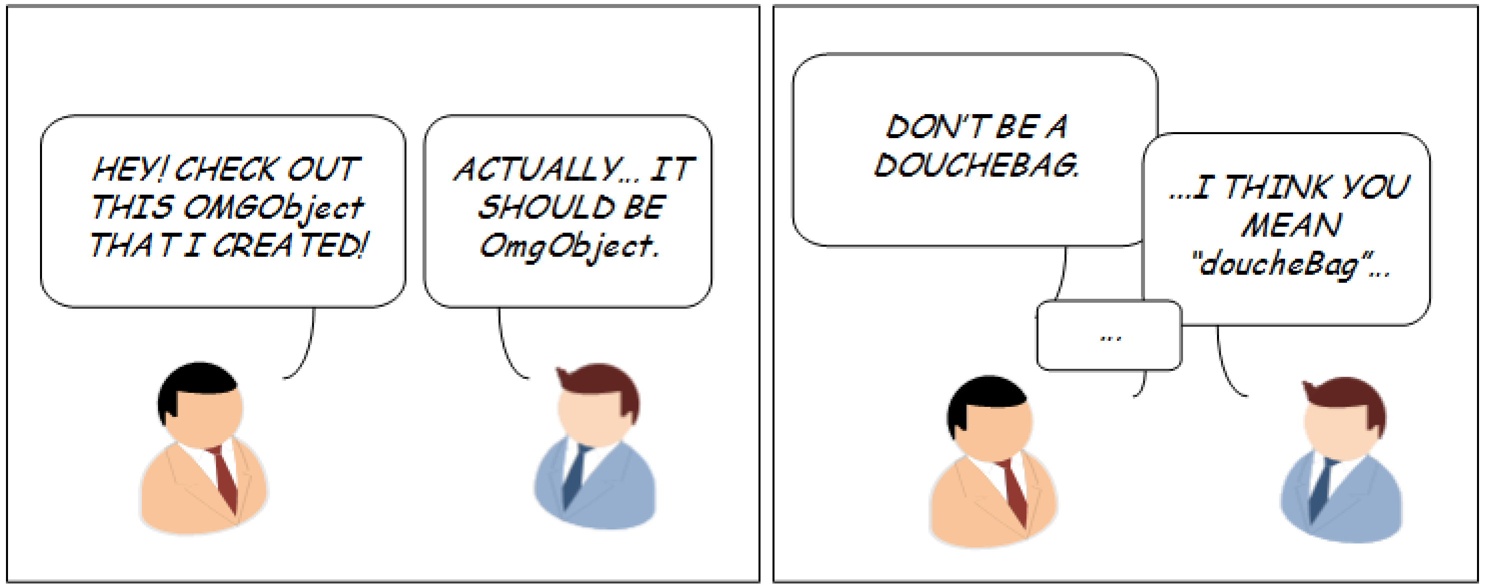
Simplicity

Reliability

Modularity

Efficient

Naming Conventions & Standard


HEY! CHECK OUT THIS OMGObject THAT I CREATED!
ACTUALLY... IT SHOULD BE OmgObject.
DON'T BE A DOUCHEBAG.
...
...I THINK YOU MEAN "doucheBag"...

Coding Best Practices

- The best practices based on the commonly made development mistakes in real time projects.
- Its address one or more of these problems such as
    - Maintainability issues
    - Performance issues

Contd....

Pascal Case

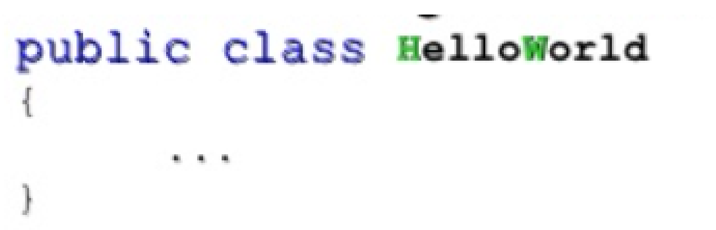
First character of all words are upper case

E.g. : BackColor

Camel Case

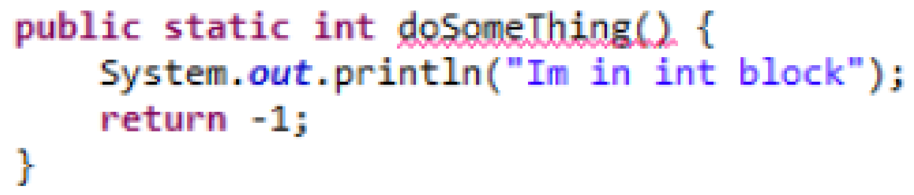
First character of all words except the first word are capitalised
E.g. : backColor

Contd....



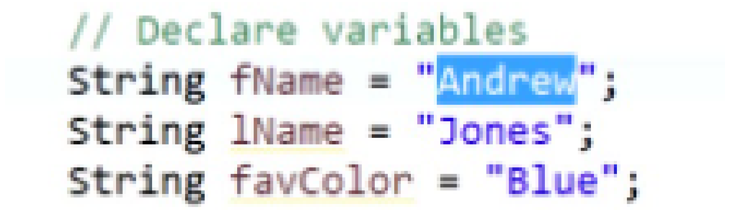
```
public class HelloWorld
{
        ...
}
```

Class names – Pascal case



```
public static int doSomeThing() {
    System.out.println("Im in int block");
    return -1;
}
```

Method name – Camel case



```
// Declare variables
String fName = "Andrew";
String lName = "Jones";
String favColor = "Blue";
```

Variables - Camel case

IEntity

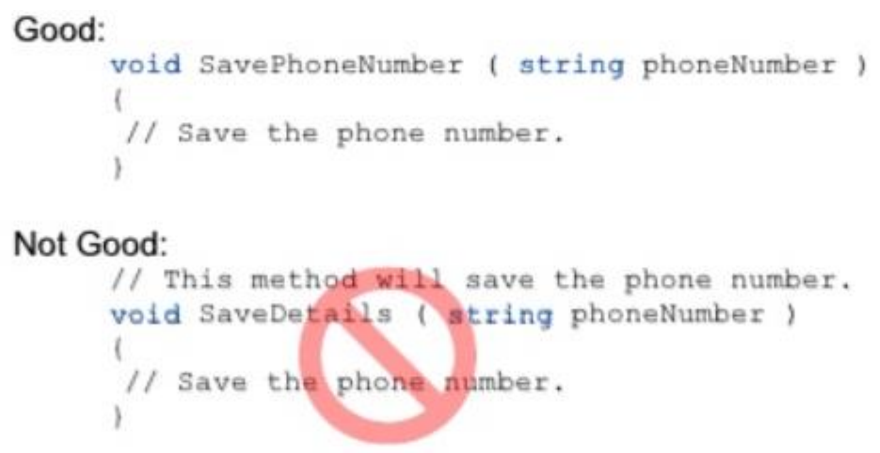
Prefix interface with I

Contd....

- Use descriptive words , do not use abbreviations
- Do not use variable names that resemble keywords.
- Prefix Boolean variables , properties and method’s with ”is” or similar prefixes . E.g. : isEvenNumber
- File name should match with class name.
- Use tab for indentation , not space

Good programming practices

- Avoid writing very long methods , if a method is too long consider refactoring it.
- Method name should tell what it does. Do not use misleading names.


Good:

```
void SavePhoneNumber ( string phoneNumber )
{
 // Save the phone number.
}
```

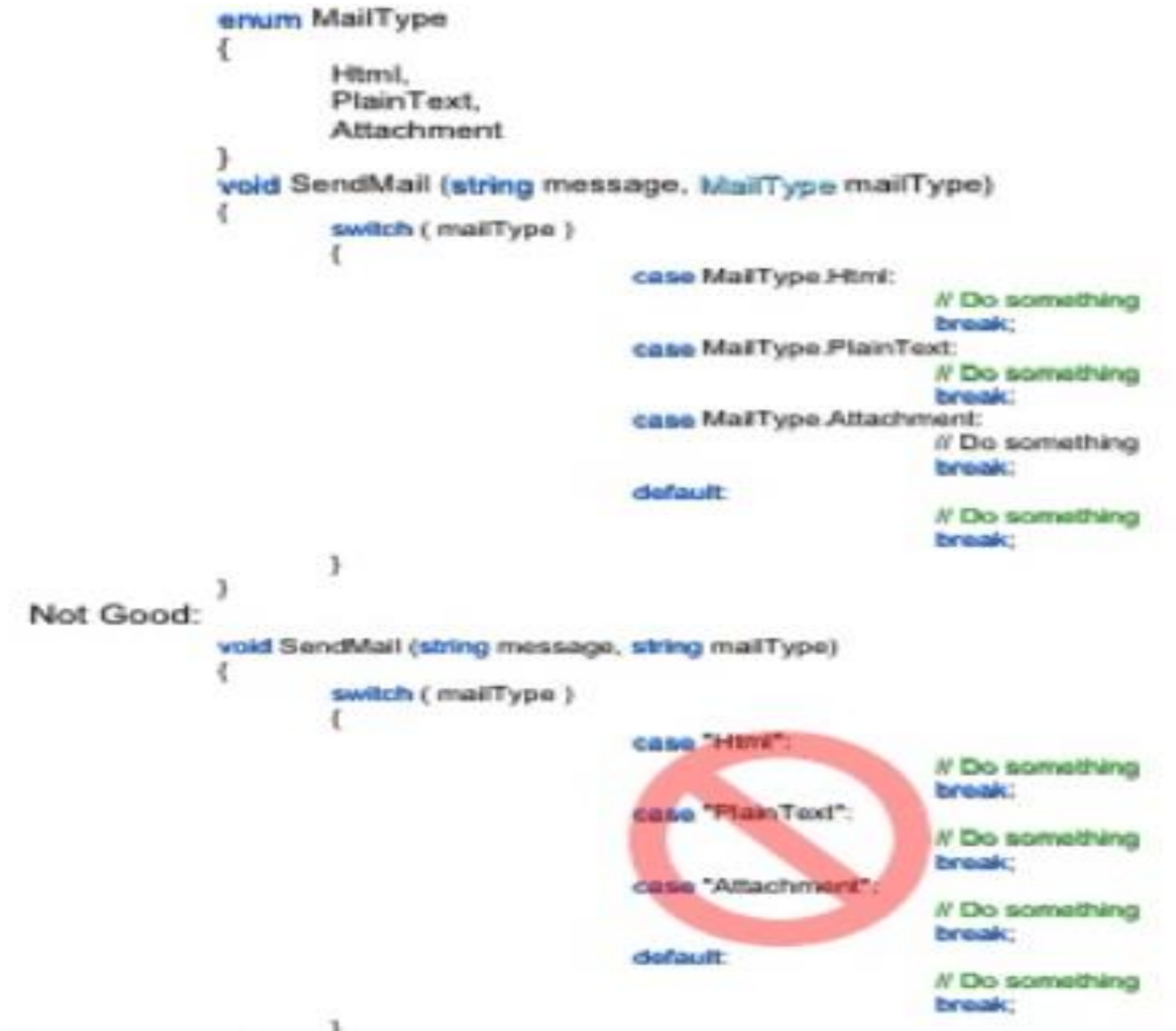
Not Good:

```
// This method will save the phone number.
void SaveDetails ( string phoneNumber )
{
 // Save the phone number.
}
```

- A method should do only one job. (Single Responsibility Principle)
- Do not hardcode numbers, use constants instead.

Good programming practices

- Do not make the member variables public or protected. Instead use protected or public getters , setters or properties
- Never hardcode a drive path.
- Use Enum , wherever required.



```
enum MailType
{
            Html,
            PlainText,
            Attachment
}
void SendMail (string message, MailType mailType)
{
            switch ( mailType )
            {
                        case MailType.Html:
                                    // Do something
                                    break;
                        case MailType.PlainText:
                                    // Do something
                                    break;
                        case MailType.Attachment:
                                    // Do something
                                    break;
                        default:
                                    // Do something
                                    break;
            }
}
```

Not Good:

```
void SendMail (string message, string mailType)
{
            switch ( mailType )
            {
                        case "Html":
                                    // Do something
                                    break;
                        case "PlainText":
                                    // Do something
                                    break;
                        case "Attachment":
                                    // Do something
                                    break;
                        default:
                                    // Do something
                                    break;
            }
```

Good programming practices

- Show short and friendly error messages to the end users. But log the error in a log file for further investigations
- If you are opening any resources such as database , file stream etc always close them in the finally block.
- Never access database from the UI layer. Always have a data layer class which performs all the database related tasks. This will help to migrate to another database easily in the future,

Documentation

- 80 % of the lifetime cost of a piece software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Good style and documentation improves the readability of the software, allowing engineers to understand new code quickly and thoroughly

Unit Testing

- Write test cases . Don’t forget the edge cases : 0s , empty strings/lists , nulls etc

Performance Tuning

- Manage data types and object lifetime
- Use string builder to concatenate strings programmatically
- Use primitive datatypes wherever possible
- Cache expensive resources e.g. : db. connection
- Optimize java heap.

Common Best Practices in Java

Few Best Practices

Quote 1: Avoid creating unnecessary objects and always prefer to do Lazy Initialization

```
public class Countries
{
                private List countries;
               public List getCountries()
              {
                //initialize only when required
                if(null == countries)
               {
                              countries = new ArrayList();
          }
         return countries;
         }
}
```

Few Best Practices cont..

Quote 2: Never make an instance fields of class public

Making a class field public can cause lot of issues in a program. For instance you may have a class called MyCalender. This class contains an array of String weekdays. You may have assume that this array will always contain 7 names of weekdays. But as this array is public, it may be accessed by anyone. Someone by mistake also may change the value and insert a bug!

```
//Wrong coding

public class MyCalender
 {
                public String[] weekdays = {"Sun", "Mon",
                "Tue", "Thu", "Fri", "Sat", "Sun"};
                //some code
}
//Best Practice
public class MyCalender
{
                private String[] weekdays = {"Sun", "Mon",
                "Tue", "Thu", "Fri", "Sat", "Sun"};
 public String[] getWeekdays()
  {
       return weekdays;
    }
}
```

Few Best Practices cont..

Quote 3: Always try to minimize Mutability of a class

Making a class immutable is to make it unmodifiable. The information the class preserve will stay as it is through out the lifetime of the class. Immutable classes are simple, they are easy to manage. They are thread safe. They makes great building blocks for other objects.

However creating immutable objects can hit performance of an app. So always choose wisely if you want your class to be immutable or not. Always try to make a small class with less fields immutable.

To make a class immutable you can define its all constructors private and then create a public static method
to initialize and object and return it.

Few Best Practices cont..

Quote 4: Try to prefer Interfaces instead of Abstract classes

First you can not inherit multiple classes in Java but you can definitely implements multiple interfaces. Its very easy to change the implementation of an existing class and add implementation of one more interface rather then changing full hierarchy of class.

Again if you are 100% sure what methods an interface will have, then only start coding that interface. As it is very difficult to add a new method in an existing interface without breaking the code that has already implemented it. On contrary a new method can be easily added in Abstract class without breaking existing functionality.

Few Best Practices cont..

Quote 5: Always try to limit the scope of Local variable

Local variables are great. But sometimes we may insert some bugs due to copy paste of old code. Minimizing the scope of a local variable makes code more readable, less error prone and also improves the maintainability of the code.

Thus, declare a variable only when needed just before its use.

Always initialize a local variable upon its declaration. If not possible at least make the local instance assigned null value.

Few Best Practices cont..

Quote 6: Try to use standard library instead of writing your own from scratch

Writing code is fun. But "do not reinvent the wheel". It is very advisable to use an existing standard library which is already tested, debugged and used by others. This not only improves the efficiency of programmer but also reduces chances of adding new bugs in your code. Also using a standard library makes code readable and maintainable.

Few Best Practices cont..

Quote 7: Wherever possible try to use Primitive types instead of Wrapper classes

Wrapper classes are great. But at same time they are slow. Primitive types are just values, whereas Wrapper classes are stores information about complete class.

```
int x = 10; int y = 10;

Integer x1 = new Integer(10);

Integer y1 = new Integer(10);

System.out.println(x == y);

System.out.println(x1 == y1);
```

Few Best Practices cont..

Quote 8: Use Strings with utmost care.

```
//slow instantiation
        String slow = new String("Yet another string object");
 //fast instantiation
        String fast = "Yet another string object";
```

Few Best Practices cont..

Quote 9: Always return empty values instead of null

Whenever your method is returning a collection element or an array, always make sure you return empty array/collection and not null. This will save a lot of if else testing for null elements. For instance in below example we have a getter method that returns employee name. If the name is null it simply return blank string “”.

```
public String getEmployeeName()
{
   return (null==employeeName ? "": employeeName);
}
```

Few Best Practices cont..

Quote 10: Never let exception come out of finally block

Finally blocks should never have code that throws exception. Always make sure finally clause does not throw exception. If you have some code in finally block that does throw exception, then log the exception properly and never let it come out

Best Practices of using Collections

- **Use Interface reference to collections:**

Don't: ✗

```
ArrayList nlist = new ArrayList();
```

Do: ✓

```
List nlist = new ArrayList();
```

Implication:

- Results in a huge development impact if the collection used needs to be changed to a different collection type.

- **Usage of collection.size() method in for loop condition:**

Don't: ✗

```
for(int i=0;i<numberList.size();i++)
{ //perform some logic
}
```

Do: ✓

```
 int size=numberList.size();
for(int i=0;i<size;i++)
{ //perform some logic  }
```

Implic

each ti

Best Practices of using Collections

- **Avoid iterating collection using index rather use iterator:**

Don't:

```
for(int idx=0;idx<10;++idx)
{ log("iteration..");   }
```

✗

Do:

```
 iterator = collection.iterator()
while(iterator.hasNext())
{
    log("iteration.."); }
```

✓

Implication:

- This is error prone and results in run time errors & index out of bounds exceptions.

- **Use collections with generics to define the data type the collection holds(JDK > 1.5):**

Don't:

```
List numberList = new ArrayList();
```

✗

Do:

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

✓

- **Implication:** Ignoring the generics will result in runtime errors. Like **ClassCastException**.

Best Practices of using Collections

- **Copy collections into other collections using add all and don't iterate them for copying:**

Don't:
Don't iterate collections to copy contents into other collections.

Do:
Use the addAll method for copying content which is efficient.

Implication:

- The results in performance degradation.

Wrapper Vs Primitive

When to use Wrapper over primitive?

- Use wrapper when you need the value stored in the wrapper needs to be manipulated such as convert integer to String or double or float etc.
- Use wrapper only if you need to store primitive values in a collection object.

Disadvantages:

- Wrapper objects eats up memory as they are objects instantiated into the java heap. Wrapper consumes 4 times the memory space if what a primitive consumes.
- CPU cycles exhausted to remove wrapper objects from memory degrading response time.

Exception Best Practices

- **Catch specific Exceptions:**
  - Be specific while handling the exception in your catch block.

  Implication:

  There are possibility that the unintended exceptions would be caught and handled, resulting in erroneous control flow.

Exception Best Practices

```
public class TaxProcessor {
        public void calculateTax(String
s1,String s) {
        try{
        calculate();
}
catch(Exception e)
{
        //handle Exception
}
Private void calculate() throws
CalculateException{
        …..
}
}
```

Don’t catch Generic Exception

```
public class TaxProcessor {
        public void calculateTax(String
s1,String s) {
        try{
        calculate();
}
catch(CalculateException e)
{
        //handle Exception
}
Private void calculate() throws
CalculateException{
        …..
}
}
```

Catch Specific Exception

Exception Best Practices

- **Exception to control program flow:**
  - Do not use Exception Handling to control programming flow.

Implication:

This makes the code tough to debug and maintain.

Don’ts:

```
public void calculateTax(String s1,String s) {
            try{
            calculateCountryTax();  //This method throws CalculateException
}
catch(CalculateException e)
{
            calculateCityTax();
}}
```

Exception controlling program flow

Exception Best Practices

- **Exception handling in loops:**
  - Do not use Exception Handling to loops, avoid it.

Implication:

This can result in loop continuing even if exception is thrown.
Resulting in eating up CPU cycles.

Don’ts:

```
public void calculateTax(String s1,String s) {
    for(int i=0;i<=10;i++) {
        try{
          calculateCountryTax();
        } catch(CalculateException e) {
               calculateCityTax();
        } } }
```

Exception handled inside loop.

Exception Best Practices

```
public void calculateTax(String s1,String s) {

        try{
            for(int i=0;i<=10;i++) {
                        calculateCountryTax();
         } catch(CalculateException e) {
                //handle Exception
         } } }
```

Exception handled outside loop.

Return Statements

- Never return values from multiple places in a method.

  Implication:

  Tough to troubleshoot code and maintain it.

```
public class ReturnBadPractice {
        private boolean validateAge(final
Person person) throws
              AgeNotValidException {
If ((peron.getAge() > 60)) || (person.getAge() >
20) && (person.getAge()<=60)) {
        return true;
} else if (person.getAge()<20) {
        return false;
} else if(person.getAge()<=0) {
        return false;
}
    return false; }}
```

```
public class ReturnBadPractice {
        private boolean validateAge(final
Person person) throws
              AgeNotValidException {
boolean eligible = false;
If ((peron.getAge() > 60)) || (person.getAge() >
20) && (person.getAge()<=60)) {
      eligible = true;
} else if (person.getAge()<20) {
        eligible  = false;
} else if(person.getAge()<=0) {
        eligible = false;
}
    return eligible;
}}
```

Loops Don’t

- Method calls, logging string concatenation inside for loop:
  - **Don’ts:**
    - Avoid DAO calls inside for loop.
    - Avoid unnecessary method calls.
    - Avoid string concatenation.
    - Avoid logging inside for loop.
    - Avoid exception handling inside loops.
  - **Implications:**
    - Response time peaks up.
    - Unnecessary object creation resulting in memory spikes or out of memory errors.

Database Best Practices

Innovative Services

Passionate Employees

Delighted Customers

How to design a database

- Normalise the data
- Define constraints to maintain data integrity
- Choose a database type (NOSQL or SQL)

Best practices for DB

- Use well defined and consistent names for tables and columns

E.g. : School , StudentCourse, CourseID

- Use singular table names

E.g. : Student Course instead of StudentCourses

- Don't use unnecessary prefixes or suffixes

E.g. : tblStudent , StudentTbl

- Keep passwords encrypted in the table, decrypt them programmatically.

- Choose columns with integer data types for indexes , strings cause performance issues.

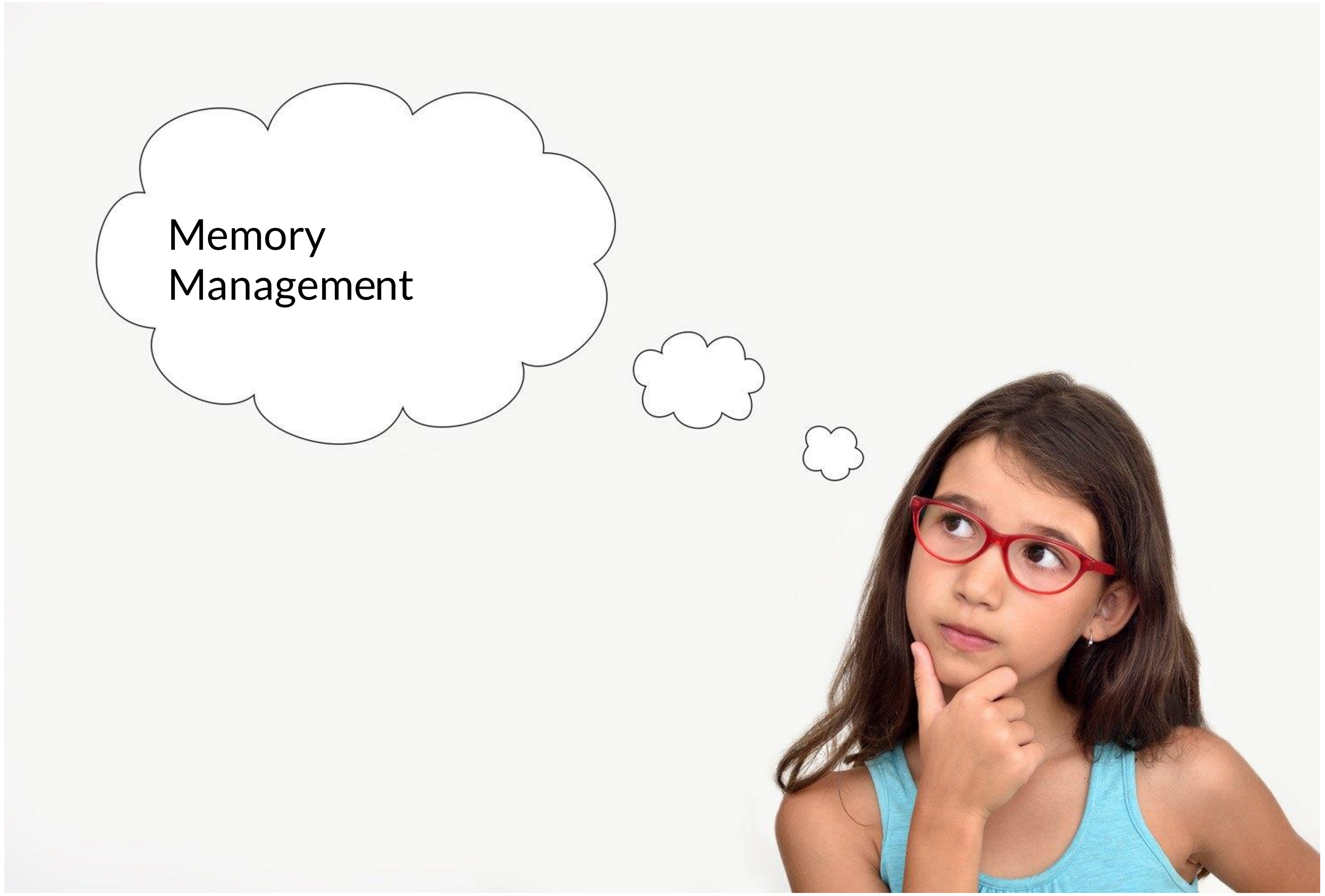
- Avoid SELECT * , instead use SELECT column names.

- Partition unused , rarely used tables parts to a different physical storage for better performance

Best practices for DB

- Use indexes on frequently used huge tables
- Image and blob must not be used in a frequently used table
- Database and Web server must be placed in different servers.
- Spend a lot of time of database modelling and design .

Memory Management

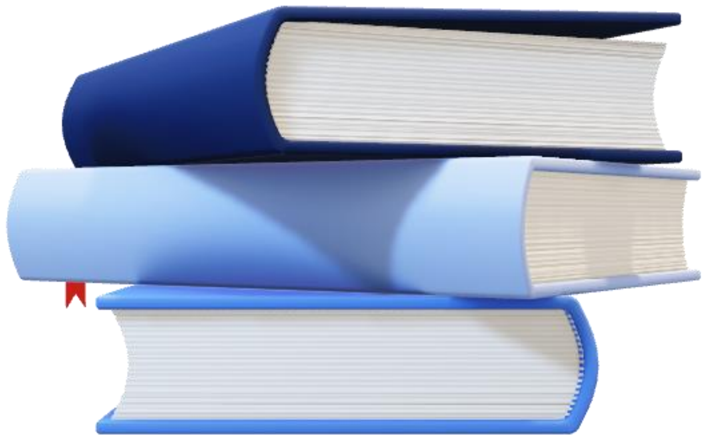

Memory
Management

Memory Management

Memory management is the process of controlling and coordinating the way a software application access **computer memory**.

How is it done?

- When a software runs on a target Operating system on a computer it needs access to the computers **RAM**(Random-access memory) to:
  - load its own **bytecode** that needs to be executed
  - store the **data values** and **data structures** used by the program that is executed
  - load any **run-time systems** that are required for the program to execute



Stack

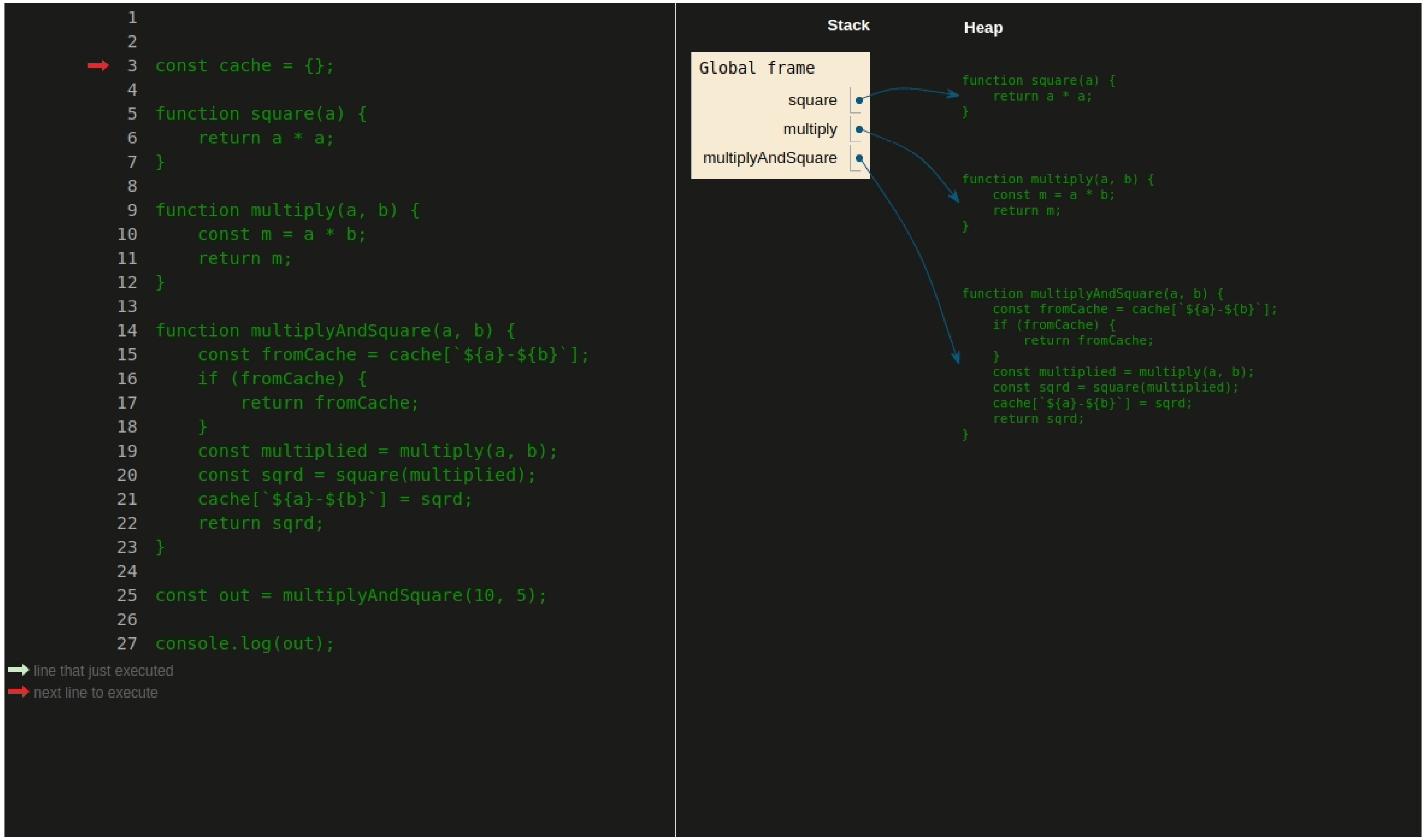
- The stack is used for **static memory allocation** and as the name suggests it is a last in first out(**LIFO**)

- The process of storing and retrieving data from the stack is **very fast** as there is no lookup required, you just store and retrieve data from the topmost block on it.

- But this means any data that is stored on the stack has to be **finite and static**(The size of the data is known at compile-time).

- This is where the execution data of the functions are stored as **stack frames**(So, this is the actual execution stack). Each frame is a block of space where the data required for that function is stored.

Contd..



```


const cache = {};

function square(a) {
    return a * a;
}

function multiply(a, b) {
    const m = a * b;
    return m;
}

function multiplyAndSquare(a, b) {
    const fromCache = cache[`${a}-${b}`];
    if (fromCache) {
        return fromCache;
    }
    const multiplied = multiply(a, b);
    const sqrd = square(multiplied);
    cache[`${a}-${b}`] = sqrd;
    return sqrd;
}

const out = multiplyAndSquare(10, 5);

console.log(out);
```

```
function square(a) {
    return a * a;
}

function multiply(a, b) {
    const m = a * b;
    return m;
}

function multiplyAndSquare(a, b) {
    const fromCache = cache[`${a}-${b}`];
    if (fromCache) {
        return fromCache;
    }
    const multiplied = multiply(a, b);
    const sqrd = square(multiplied);
    cache[`${a}-${b}`] = sqrd;
    return sqrd;
}
```

Contd…

- **Multi-threaded applications** can have a **stack per thread**.
- Memory management of the stack is **simple and straightforward** and is done by the OS.
- Typical data that are stored on stack are **local variables**(value types or primitives, primitive constants), **pointers** and **function frames**.
- This is where you would encounter **stack overflow errors** as the size of the stack is limited compared to the Heap.
- There is a **limit on the size** of value that can be stored on the Stack for most languages.

Heap

- Heap is used for **dynamic memory allocation** and unlike stack, the program needs to look up the data in heap using **pointers** (Think of it as a big multi-level library).

- It is **slower** than stack as the process of looking up data is more involved but it can store more data than the stack.

- This means data with **dynamic size** can be stored here.

- Heap is **shared** among threads of an application.

Contd..

- Due to its dynamic nature heap is **trickier to manage** and this is where most of the memory management issues arise from and this is where the automatic memory management solutions from the language kick in.
- Typical data that are stored on the heap are **global variables**, **reference types** like objects, strings, maps, and other complex data structures.
- This is where you would encounter **out of memory errors** if your application tries to use more memory than the allocated heap(Though there are many other factors at play here like GC, compacting).
- Generally, there is **no limit** on the size of the value that can be stored on the heap. Of course, there is the upper limit of how much memory is allocated to the application.

Why should memory be managed

Unlike Hard disk drives, RAM is not infinite.

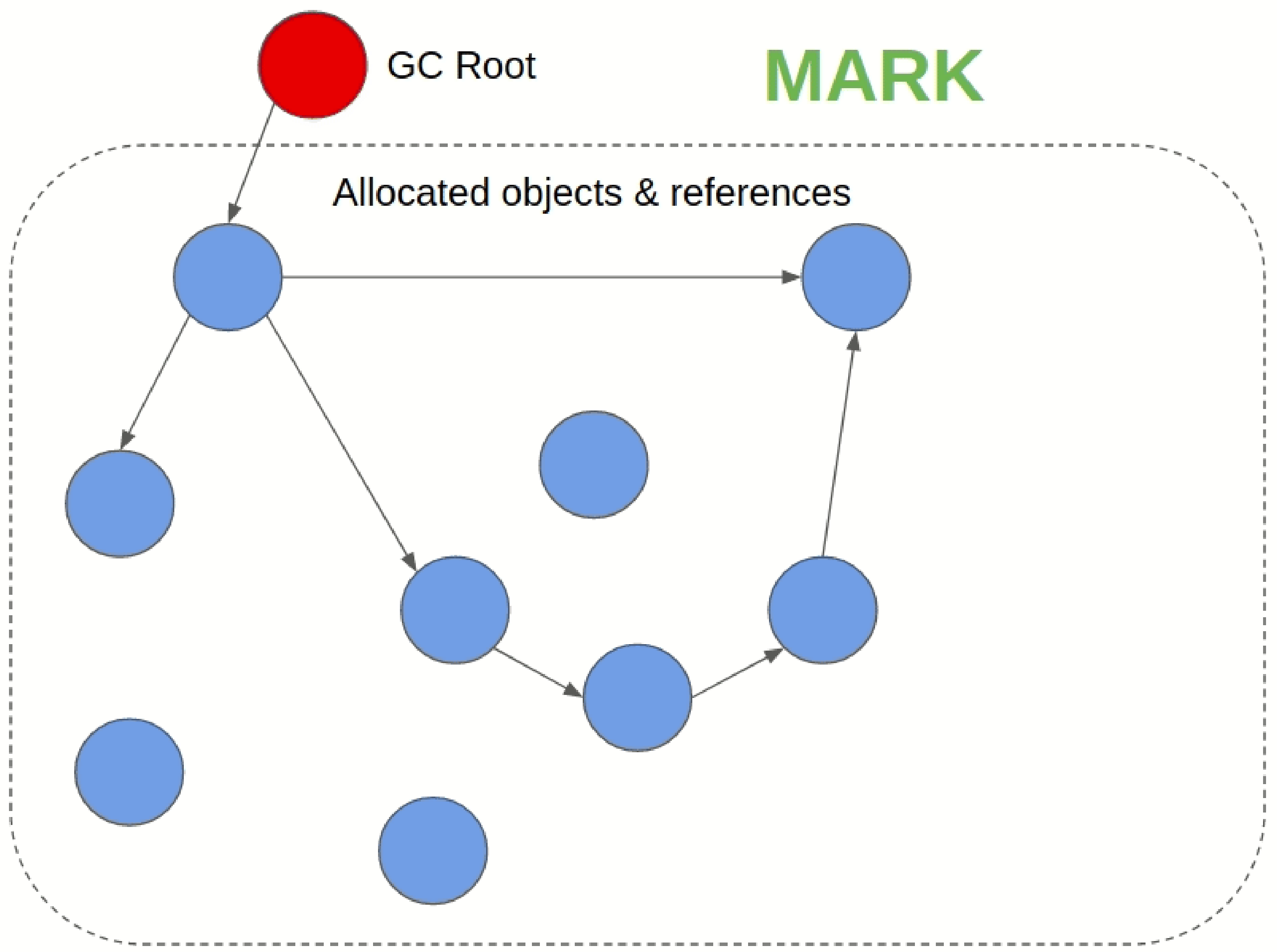
Garbage Collection

- Automatic management of heap memory by freeing unused memory allocations.

- GC is one of the most common memory management in modern languages and the process often runs at certain intervals and thus might cause a minor overhead called pause times.

- **JVM(Java/Scala/Groovy/Kotlin)**, **JavaScript**, **C#**, **Golang**, **OCaml**, and **Ruby** are some of the languages that use Garbage collection for memory management by default.

Garbage Collection



Mark & Sweep GC

- Its generally a two-phase algorithm that first marks objects that are still being referenced as “alive” and in the next phase frees the memory of objects that are not alive.

- **JVM**, **C#**, **Ruby**, **JavaScript**, and **Golang** employ this approach for example.

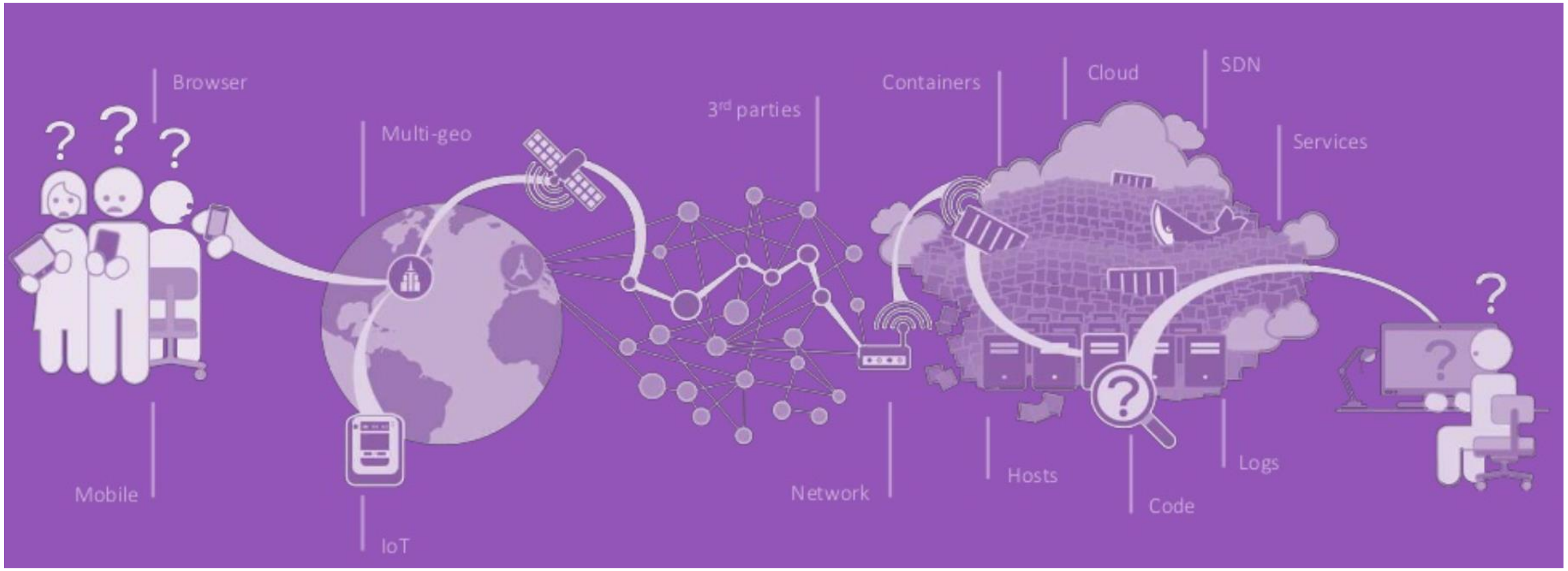
- In JVM there are different GC algorithms to choose from while JavaScript engines like V8 use a Mark & Sweep GC along with Reference counting GC to complement it. This kind of GC is also available for C & C++ as an external library.

Reference counting GC

- In this approach, every object gets a reference count which is incremented or decremented as references to it change and garbage collection is done when the count becomes zero. It’s not very preferred as it cannot handle cyclic references. **PHP**, **Perl Python**, for example, uses this type of GC with workarounds to overcome cyclic references. This type of GC can be enabled for C++ as well.

Self Healing

Self Healing



Applications are getting more complex!! A single application might use up to 82 different technology.

Self Healing

- In information technology , self healing describes any device or system that has the ability to understand that it is not operating correctly and without human intervention, make the adjustments to restore itself to normal operations.
- A self healing software system is one that has the ability to discover , diagnose and repair .
- A workflow which triggers and responds to alerts or events by executing actions that can prevent or fix the problem.
- E.g. : restart a service when it is down and create a ticket to do root cause analysis in business hours.

Characteristics of a self healing system

- System must know itself
- System must be able to reconfigure itself within its operational environment
- System must pre-emptively optimise itself.
- System must detect and respond to its own fault as they develop
- System must detect and respond to intrusions and attacks
- System must know its context to use
- System must frequently shrink the gap between user goals and IT solutions.

What is needed for self – healing applications

MONITORING

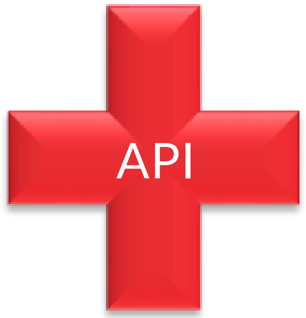
Know what is going on in your app

- End – to end
- Full stack – fully integrated in production

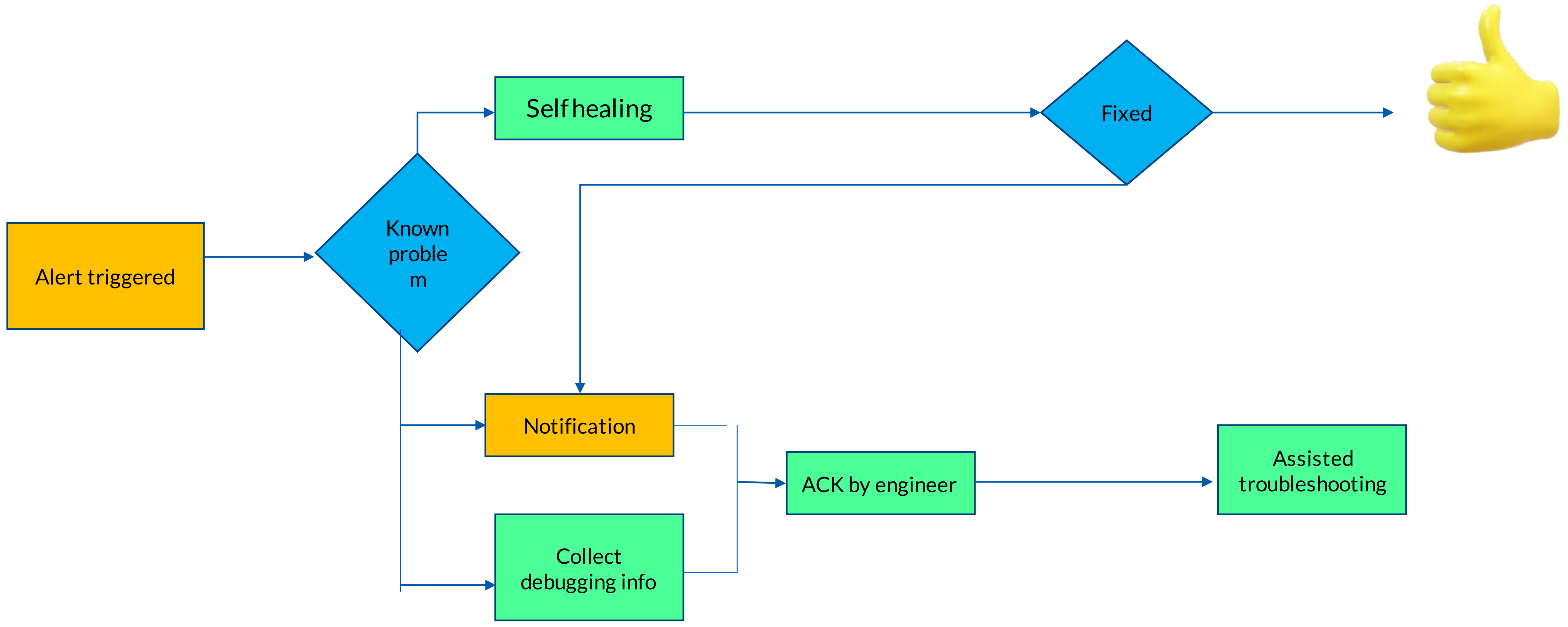
AUTOMATION/MITIGATION

Perform mitigation actions

- Access to all systems
- Automation system must be isolated from production system



Contd…


Alert triggered
Known proble m
Self healing
Fixed
Notification
Collect debugging info
ACK by engineer
Assisted troubleshooting

Steps of Self Healing applications

- Detect failure
- Notify "the system " of failure event
- The system listens for failure events and triggers auto healing response
- Logging of all actions and attempts to heal ; to monitor and improve 'the system'

Checkstyle

Checkstyle

- Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard.
- It automates the process of checking Java code.
- This makes it ideal for projects that want to enforce a coding standard.
- Checkstyle is highly configurable and can be made to support almost any coding standard.

Checks - JavadocMethod

- Checks the Javadoc of a method or constructor.
- Violates parameters and type parameters for which no param tags are present can be suppressed by defining property allowMissingParamTags.
- Violates methods which return non-void but for which no return tag is present can be suppressed by defining property allowMissingReturnTag.
- Violates exceptions which are declared to be thrown but for which no throws tag is present by activation of property validateThrows.

```
<module name="JavadocMethod"/>
```

Checks - JavadocType

- Checks the Javadoc comments for type definitions.
- By default, does not check for author or version tags.
- The scope to verify is specified using the Scope class and defaults to Scope.PRIVATE.
- To verify another scope, set property scope to one of the Scope constants.

```
<module name="JavadocType"/>
```

Checks – JavadocVariable Contd…

- Checks that a variable has a Javadoc comment.

```
<module name="JavadocVariable">
  <property name="scope" value="public"/>
</module>
```

- Specify the visibility scope where Javadoc comments are checked.

Checks – JavadocVariable

Example:

```
public class Test {
private int a; // violation, missing javadoc for private member

/**
 * Some description here
 */
private int b; // OK
protected int c; // OK
public int d; // OK
/*package*/ int e; // violation, missing javadoc for package member
}
```

Checks - LineLength

- Checks for long lines.
- To configure the check to accept lines up to 120 characters long

```
<module name="LineLength">
  <property name="max" value="120"/>
</module>
```

Checks - ParameterNumber

- Checks the number of parameters of a method or constructor.

name	description	type	default value
max	Specify the maximum number of parameters allowed.	int	7

- To configure the check to allow 10 parameters for a method:

```
<module name="ParameterNumber">
  <property name="max" value="10"/>
  <property name="tokens" value="METHOD_DEF"/>
</module>
```

Checks - MethodLength

- Checks for long methods and constructors.
- To configure the check so that it accepts methods with at most 60 lines:

```
<module name="MethodLength">
  <property name="tokens" value="METHOD_DEF"/>
  <property name="max" value="60"/>
</module>
```

Checks – GenericWhitespace Contd…

- Checks that the whitespace around the Generic tokens (angle brackets) "<" and ">" are correct to the typical convention. The convention is not configurable.

Left angle bracket ("<")

- should be preceded with whitespace only in generic methods definitions.
- should not be preceded with whitespace when it is preced method name or constructor.
- should not be preceded with whitespace when following type name.
- should not be followed with whitespace in all cases.

Checks – GenericWhitespace

Right angle bracket (">"):

- should not be preceded with whitespace in all cases.
- should be followed with whitespace in almost all cases, except diamond operators and when preceding method name or constructor.
- <module name="GenericWhitespace"/>

Checks – NoWhitespaceAfter

- Checks that there is no whitespace after a token.
- More specifically, it checks that it is not followed by whitespace, or (if linebreaks are allowed) all characters on the line after are whitespace.
- To forbid linebreaks after a token, set property allowLineBreaks to false.

```
<module name="NoWhitespaceAfter"/>
```

Checks – NoWhitespaceBefore

- Checks that there is no whitespace before a token.
- More specifically, it checks that it is not preceded with whitespace, or (if linebreaks are allowed) all characters on the line before are whitespace.
- To allow linebreaks before a token, set property allowLineBreaks to true. No check occurs before semi-colons in empty for loop initializers or conditions.

```
<module name="NoWhitespaceBefore"/>
```

Checks – WhitespaceAfter & WhitespaceAround

- Checks that a token is followed by whitespace, with the exception that it does not check for whitespace after the semicolon of an empty for iterator.

  <module name="WhitespaceAfter"/>

- Checks that a token is surrounded by whitespace. Empty constructor, method, class, enum, interface, loop bodies (blocks), lambdas of the form

Checks – Indentation Contd…

- Checks correct indentation of Java code.
  <module name="Indentation"/>

- Basic offset indentation is used for indentation inside code blocks.

- For any lines that span more than 1, line wrapping indentation is used for those lines after the first.

- Brace adjustment, case, and throws indentations are all used only if those specific identifiers start the line.

- If, for example, a brace is used in the middle of the line, its indentation will not take effect. All indentations have an accumulative/recursive effect when they are triggered. If during a line wrapping, another code block is found and it doesn't end on that same line, then the subsequent lines afterwards, in that new code block, are increased on top of the line wrap and any indentations above it.

Checks - Indentation

name	description	type	default value
basicOffset	Specify how far new indentation level should be indented when on the next line.	int	4
braceAdjustment	Specify how far a braces should be indented when on the next line.	int	0
caseIndent	Specify how far a case label should be indented when on next line.	int	4
throwsIndent	Specify how far a throws clause should be indented when on next line.	int	4
arrayInitIndent	Specify how far an array initialisation should be indented when on next line.	int	4
lineWrappingIndentation	Specify how far continuation line should be indented when line-wrapping is present.	int	4

s

- f

Thank you

Innovative Services

Passionate Employees

Delighted Customers