



TECHPROED

PROFESSIONAL TECHNOLOGY EDUCATION

WELCOME TO TECHPROED JAVA TUTORIAL

Fall-2021 Batch 49 - 54

Reminding...

1. Attendance



2. Listen to the instructor carefully & Be active listener



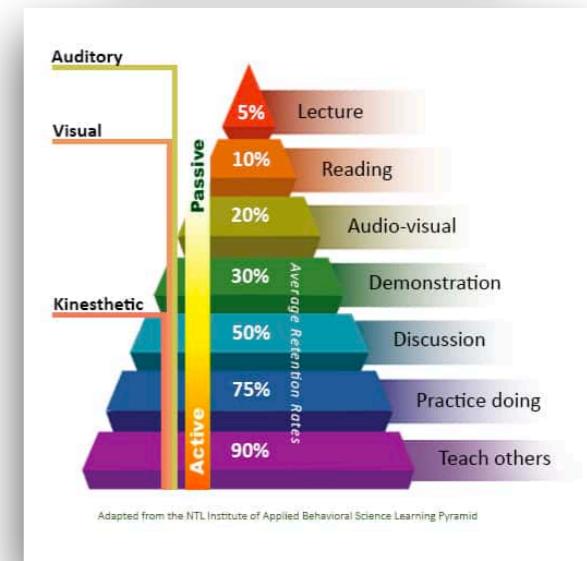
3. Daily reviews

4. Do assignments

5. Teach more, learn more

6. Bootcamps need to study hard.

Success = Instruction + Study



7. Mentoring is essential

8. Be positive, if you have any problem please reach out to (+1 917 768 74 66)

9. For technical support please reach out to Mr. Abdullah through “slack” ([@technical support](#))



10. Before every class, make sure your applications are up and running.

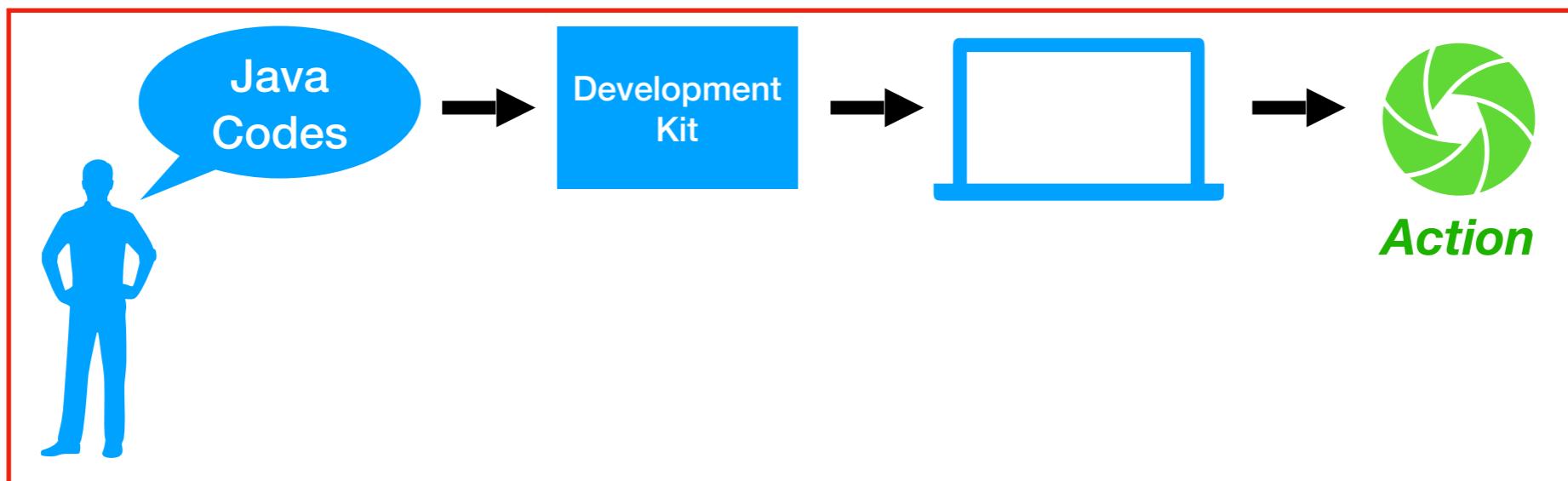
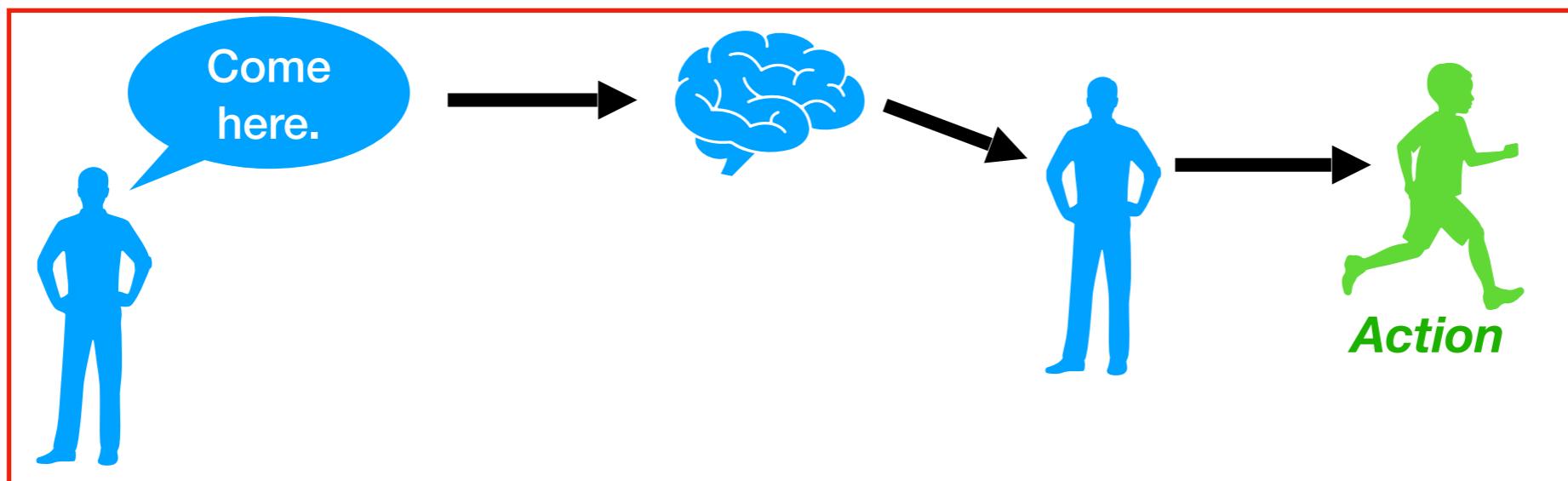
11. Check your email and slack messages **at least once** in a day

12. How to use “slack”



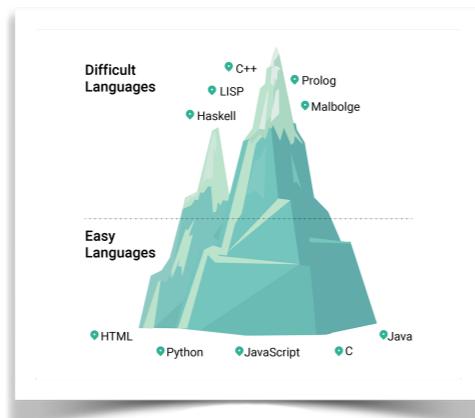
13. How to create “Bookmark”

What is Programming Language ?



Why Java ?

1. Easy to learn

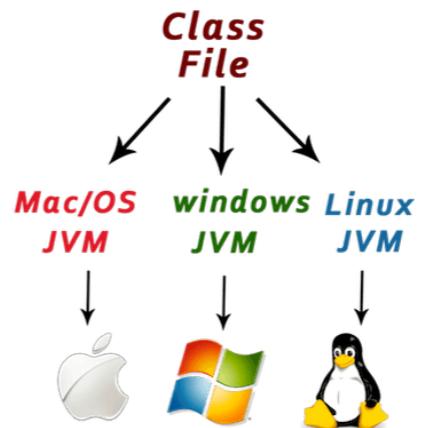


2. It's the most commonly used programming language in the world

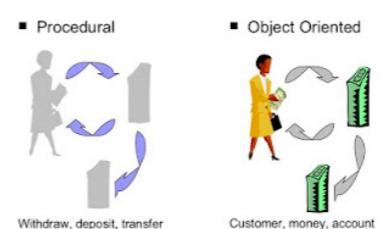
Which programming language is best for getting job in USA?

- JAVA. For many years, JAVA has been one of the most demanding **programming languages** ...
- Python. Python, named after Monty Python, is the leading player in the **programming world** ...
- C Language. C language can be considered as the core of **programming languages** as almost all low-level systems like operating systems, etc. ...
- Swift. ...
- PHP.

3. Java is a Platform Independent Programming Language



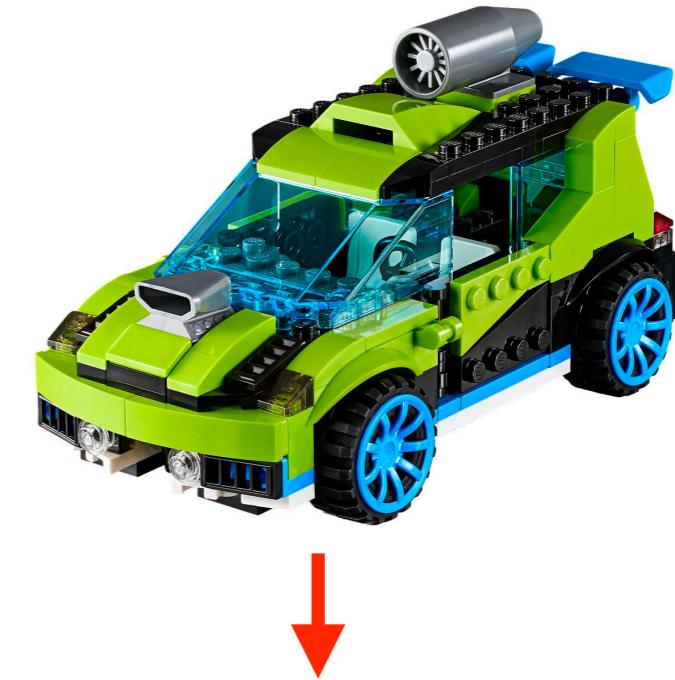
Procedural vs. Object-Oriented



4. Java is an Object Oriented Programming Language



Object Oriented Programming Language (OOP)



Object



- 1) Feature (*Field - Variable*)
- 2) Functionality (*Method*)

Application



How to Create an Object ?



Class

Variables
(Fields)

↓
Size

Color

Methods
(Functions)

↓
Buy

Travel

Object

↓
Many Objects

↓
Integrate Objects

↓
Application



```
public class BeveragePage {  
  
    private String beveragePageTitleXpath = "//h1[text() = 'Beverage']";  
    private String buyNowButtonXpath = "//a[contains(text(), 'Buy Now')]";  
    private String searchBarXpath = "//input[@id='searchByCode']";
```

→ **Field / Variables**

```
@Step("Verify to be on beverage page")  
public boolean isOnBeveragePage() {  
    return $x(beveragePageTitleXpath).shouldHave(Condition.text("Beverage")).isDisplayed();  
}
```

→ **Method**

```
@Step("Select a specific product")  
public GuestSelectionPage selectAProductByCode(ProductDetails productDetails) {  
    $x(searchBarXpath).sendKeys(productDetails.getVariantDetails().getVariantCode());  
    $x(searchBarXpath).pressEnter();  
    $x(buyNowButtonXpath).click();  
    return new GuestSelectionPage();  
}
```

→ **Method**



Keywords in a Class

```
public class MyFirstClass {}
```

public is an access modifier

An access modifier **restricts the access** of a class

Other access modifiers:

protected, default, private

MyFirstClass is class name, start with **uppercase** to name it

The part starts with “**{**” and ends with “**}**” is called **body**



How to create a method ?

```
public int myFirstMethod () { }
```

public is an access modifier

An access modifier **restricts** the **access** of a method

int is return type

return type is the type of the data which method produces

myFirstMethod() is the method name starts with **lowercase**,
the other words start with **uppercase**

The part starts with “{” and ends with “}” is called **method body**

Main Method

Main Method is the **Entry Point** of any java program.



Car → Engine

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

Java Project → Main Method



How to add comments among the codes ?

1) One line comment

// After double forward slash type your comment

2) Multi-line comments

Between /* and */ type your multi line comments

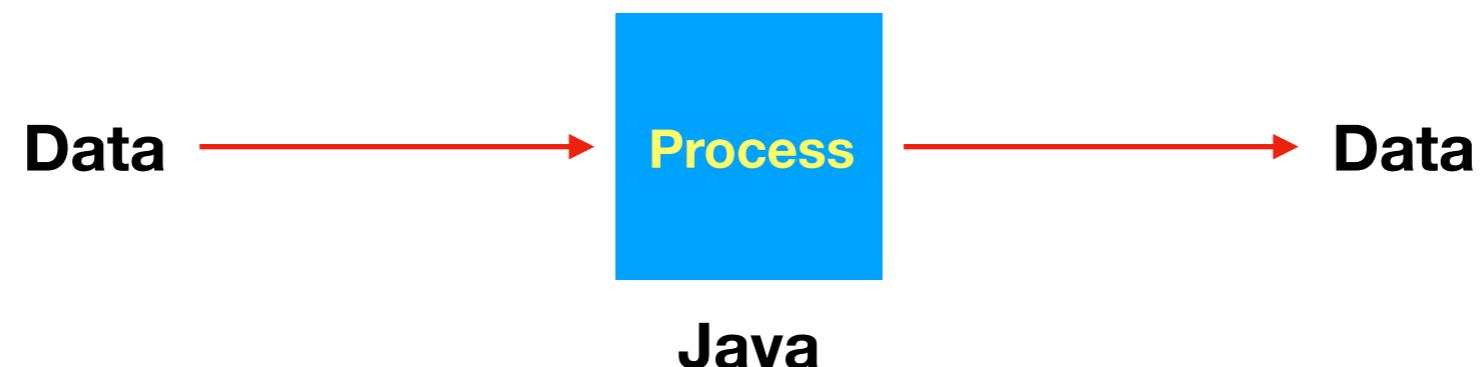
```
/*
Comment line 1
Comment line 2
Comment line 3
*/
```



Data

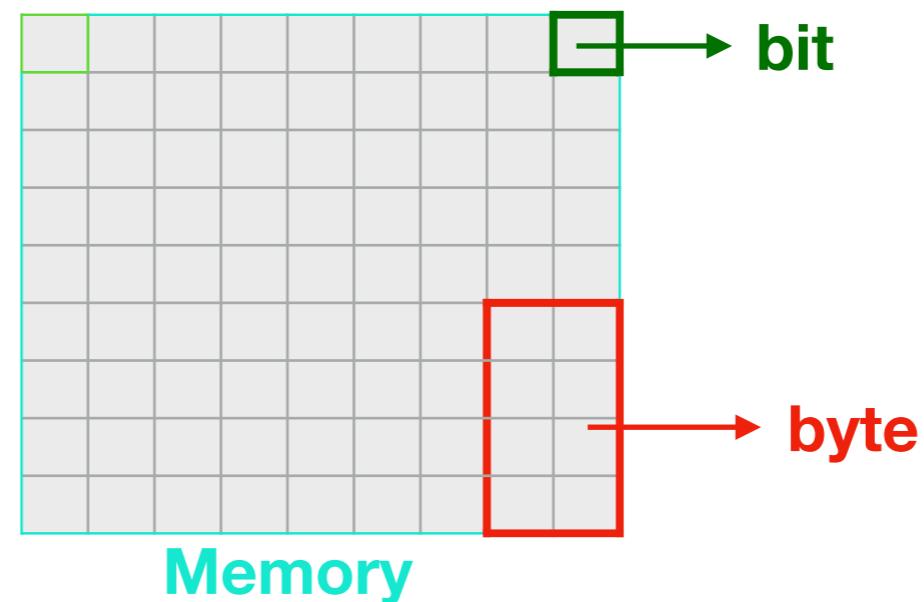
Data is information **processed** or **stored** by a computer.

Everything **java uses** or everything **java produces** is **data**



Bit

A bit is the **smallest unit of data** in a computer.
A bit has a single binary value, either 0 or 1.



Note: 8 bits are called 1 byte



How to Create a Package

Right click on the "src"

Select "New"

Select "Package"

*Type “**day2variablesscanner**” as package name*

Click on “Finish”

How to Create a Class

Right click on the “day2variables” package

Select “New”

Select “Class”

*Type “**Variables01**” as class name*

Click on “Finish”



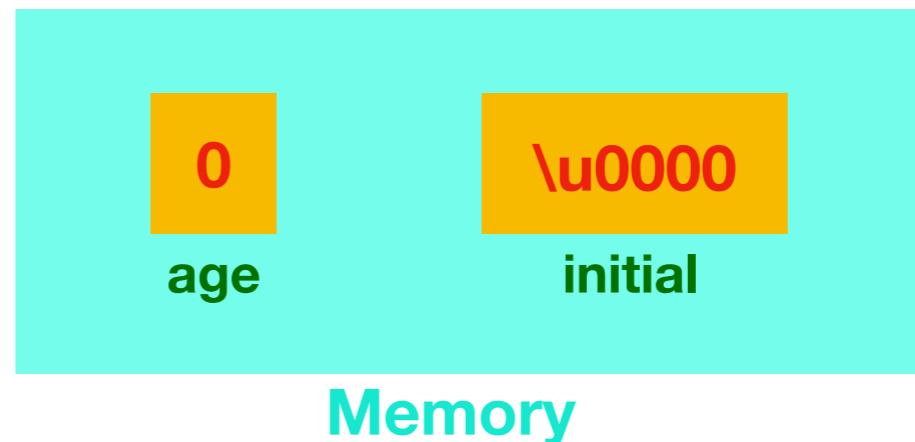
Java Variables

Variable is the name of **reserved area** allocated in memory. In other words, it is a name of **memory location**. A variable is a **container** which holds the value while the java program is executed.

Variable Declaration :

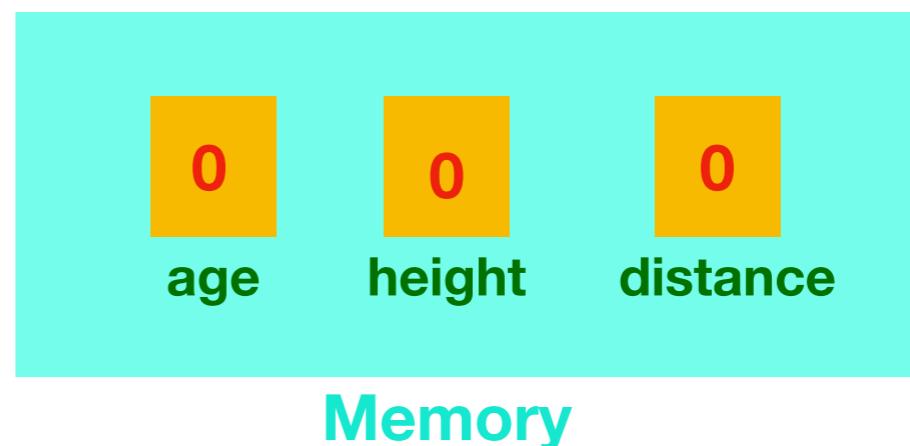
Data Type VariableName ;

int age ;
char initial ;



Note: If you will declare **more than one** variable with the **same data type**;

Data Type VariableNames ;
int age, height, distance ;



How to assign a value to a variable:

Data Type VariableName = Value;

int age = 27 ;

char initial = 'A' ;



Memory

Note: Declaration and assignment can be done in different ways like;

```
int age;  
age = 27 ;
```

27

Memory

```
int age, height, distance ;  
age = 27 ;  
height = 185 ;  
distance = 470 ;
```

27

185

470

age

height

distance

Memory

```
int age=27, height=185, distance=470 ;
```

27

185

470

age

height

distance

Memory

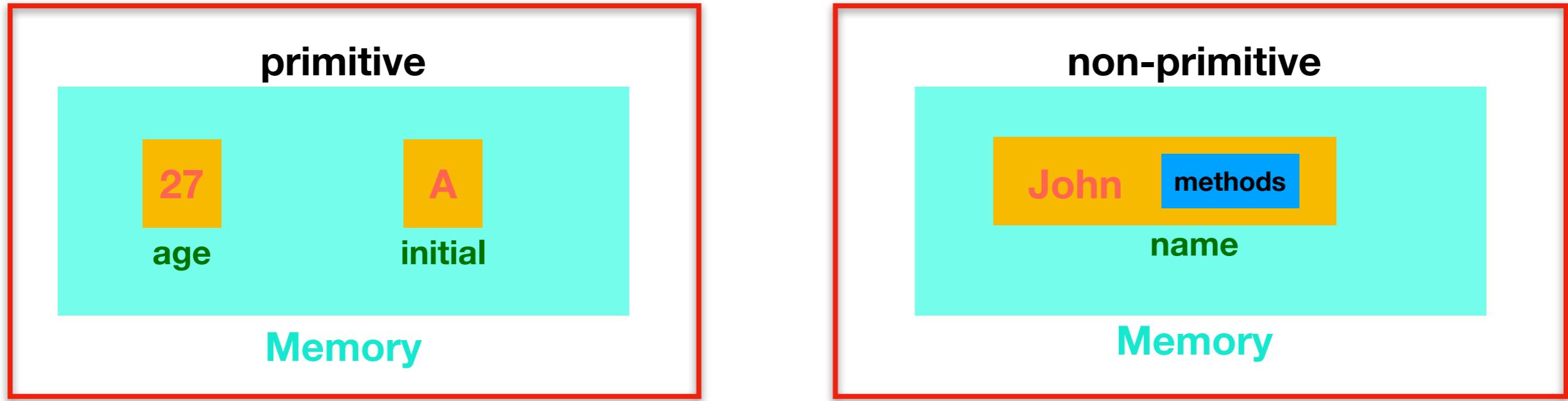


Data Types in Java

There are two types of **data types** in Java:

- Primitive data types:** The primitive data types include **boolean**, **char**, **byte**, **short**, **int**, **long**, **float** and **double**.
- Non-primitive data types:** The non-primitive data types include **String**.

Difference between “primitive” and “non-primitive” data types



- 1) Primitives contain **just values**, non-primitives contain **values and methods** inside the container.
- 2) Primitives start with **lower case**, non-primitives start with **upper case**.
- 3) Primitives are **created by Java**, we cannot create primitive data types.
Non-primitives can be created by programmer, java created some as **String**.
- 4) The size of primitive types depend on the data type, non-primitives types have all the **same size**.

Primitive Data Types

boolean Data Type: They are used to store only two possible values which are **true** and **false**. **Represents only 1 bit.**

```
boolean isExpensive = true;
```

```
boolean isCold = false;
```

char Data Type: It stores a **single character** and generally requires a single byte of memory. **Represents only 16 bit**

```
char letter = 'a';
```

```
char digit = '3';
```

Note: Put the char value between single quotes

byte Data Type: Its value-range lies between **-128 to 127** (inclusive). **Represents only 8 bit**

```
byte age = 73;
```

```
byte heightOfBuilding = 112;
```

short Data Type: Its value-range lies between **-32,768 to 32,767** (inclusive). **Represents only 16 bit**

```
short populationOfTown = 27,324;
```

```
short lossOfCompany = -15,675;
```



int Data Type: Its value-range lies between **-2,147,483,648** to **2,147,483,647** (inclusive). **Represents only 32 bit**

```
int profitOfApple = $1,342,345,000;
```

```
int lossOfNokia = -1,125,675,765;
```

long Data Type: Its value-range lies between **-9,223,372,036,854,755,808** to **9,223,372,036,854,755,807** (inclusive).

Represents only 64 bit

```
long profitOfApple = $1,342,345,000;
```

```
long lossOfNokia = -1,125,675,765;
```

float Data Type: The float data type is up to 7 decimal digits. **Represents only 32 bit**

```
float floatVar1 = 2.123 f;
```

```
float floatVar2 = -2.1 2 3 4 5 6 f;
```

Note: Put “f” at the end of the float value, otherwise it is accepted as double

double Data Type: The float data type is up to 16 decimal digits. **Represents only 64 bit**

```
double doubleVar1 = 2.123;
```

```
double doubleVar2 = -2.1234567907800000000123;
```



ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	:	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	-
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(88	58	1011000	130	X					
41	29	101001	51)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135]					
46	2E	101110	56	,	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Hexadecimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A, B, C, D, E, F

Octal: 0, 1, 2, 3, 4, 5, 6, 7 TECH PRO ED



Non - Primitive Data Types

String Data Type: String is a sequence of connected characters

String nameOfSchool = “Dade College”;

String characters = “12!/?<;

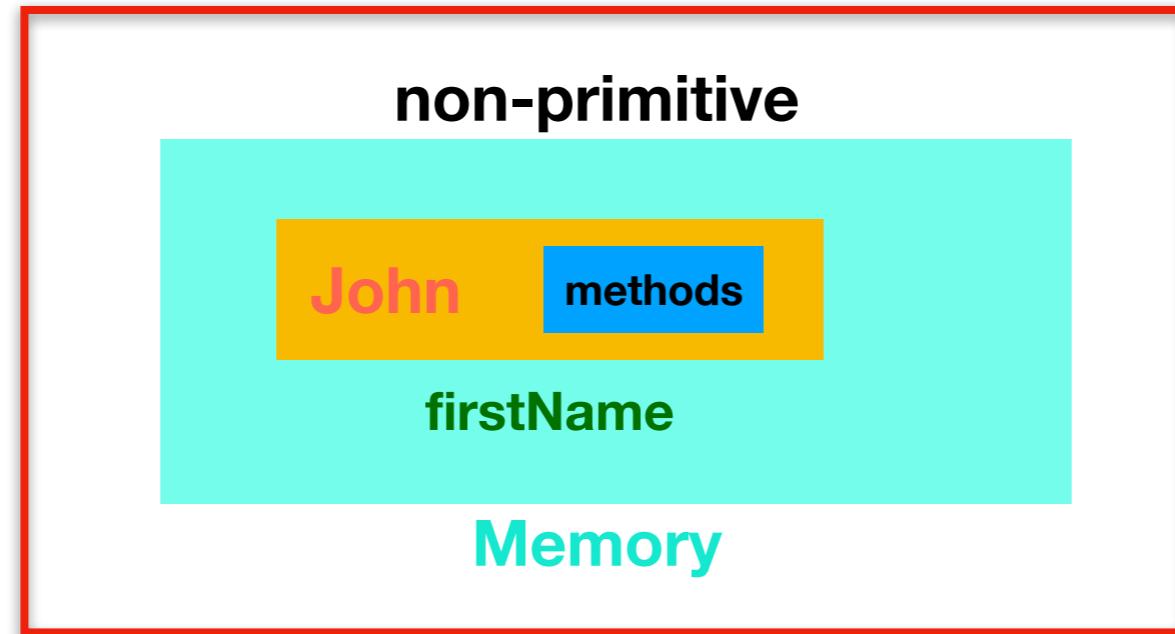
String characters = “1”;

Note: Put the String value between double quotes

Note: There are other non-primitive data types, we will learn them later



String



How to Create a String

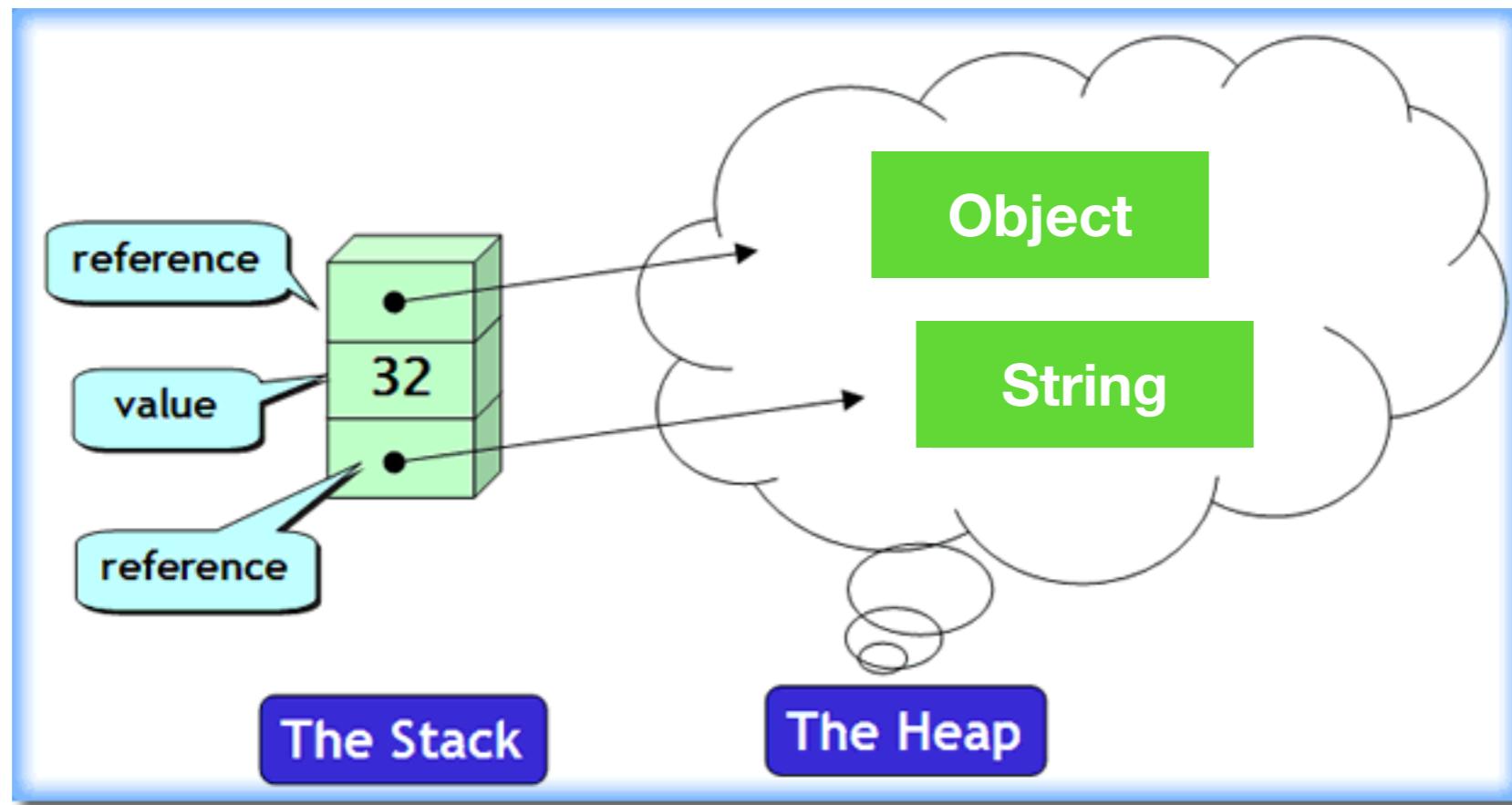
Data Type VariableName = Value;

String firstName = “John” ;



Memory Usage in Java

There are two types of memory in Java, one is **Stack**, the other is **Heap**



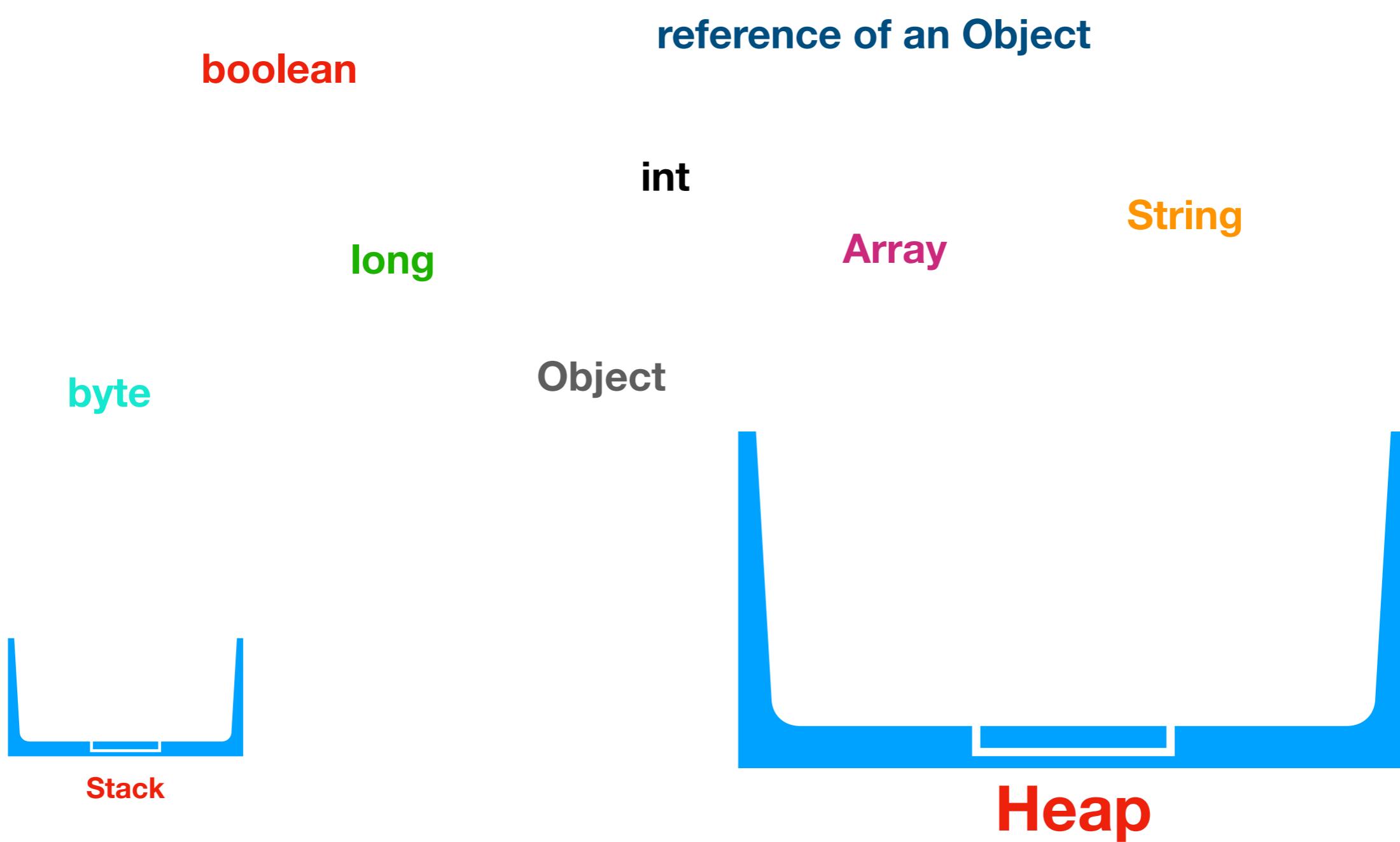
Stack memory contains **primitive** data type **values** and **references** for **objects**

Note : Non - Primitive data types are object

Heap memory is used to store **objects** in Java

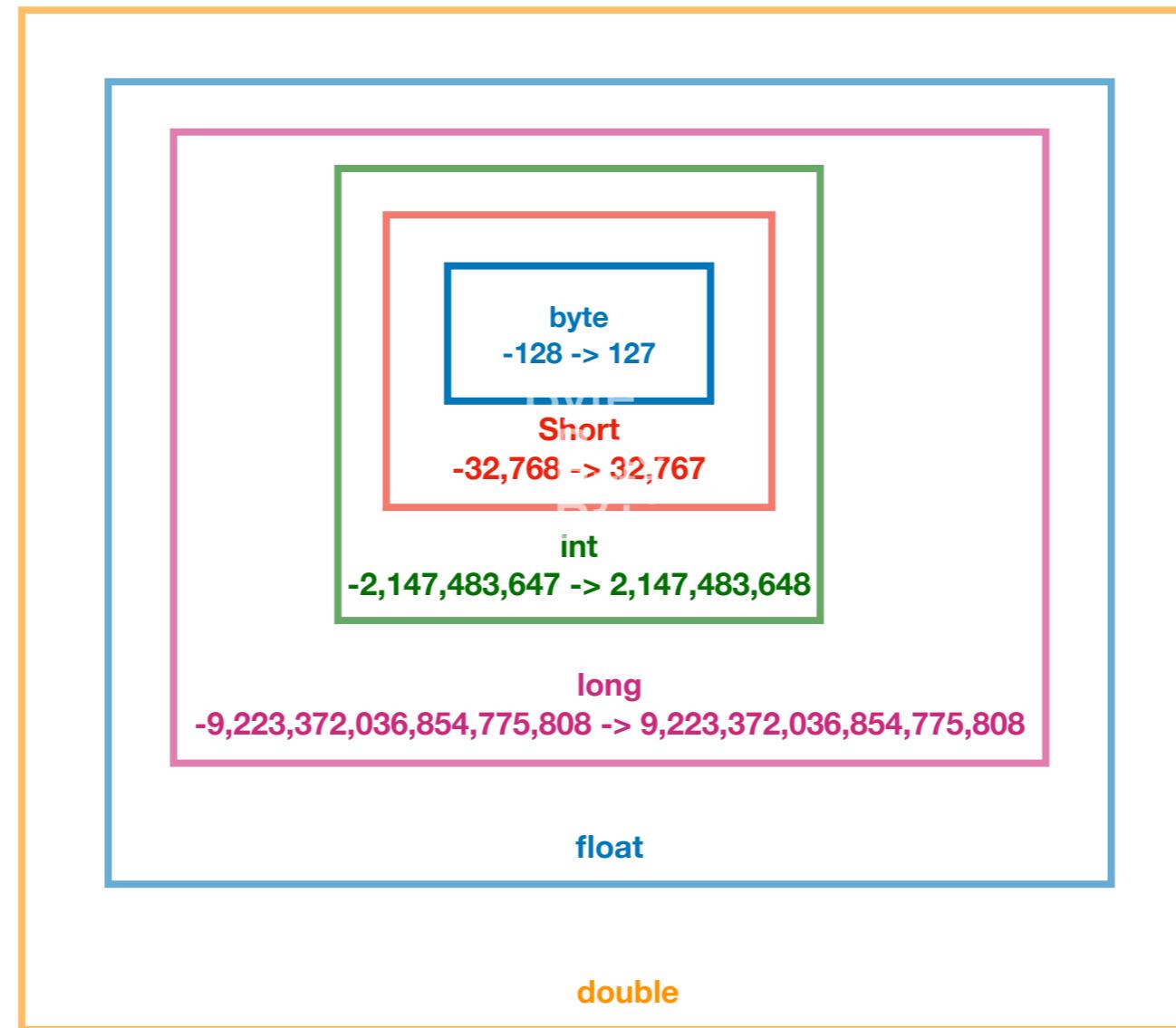
Stack is very small according to the Heap

Example: Put the followings inside the appropriate memory

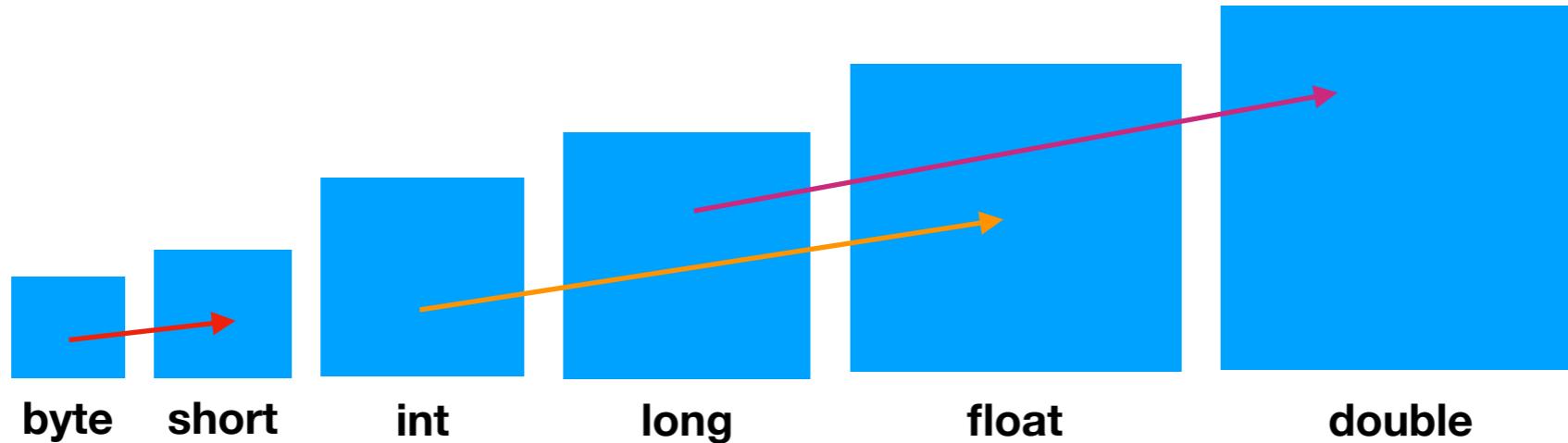


Data Type Casting

If you assign a value of one primitive data type to another type, it is called “Type Casting”



1) Auto Widening Casting: If you assign a smaller data type to a larger data type then java converts the data type **automatically** to the larger one.

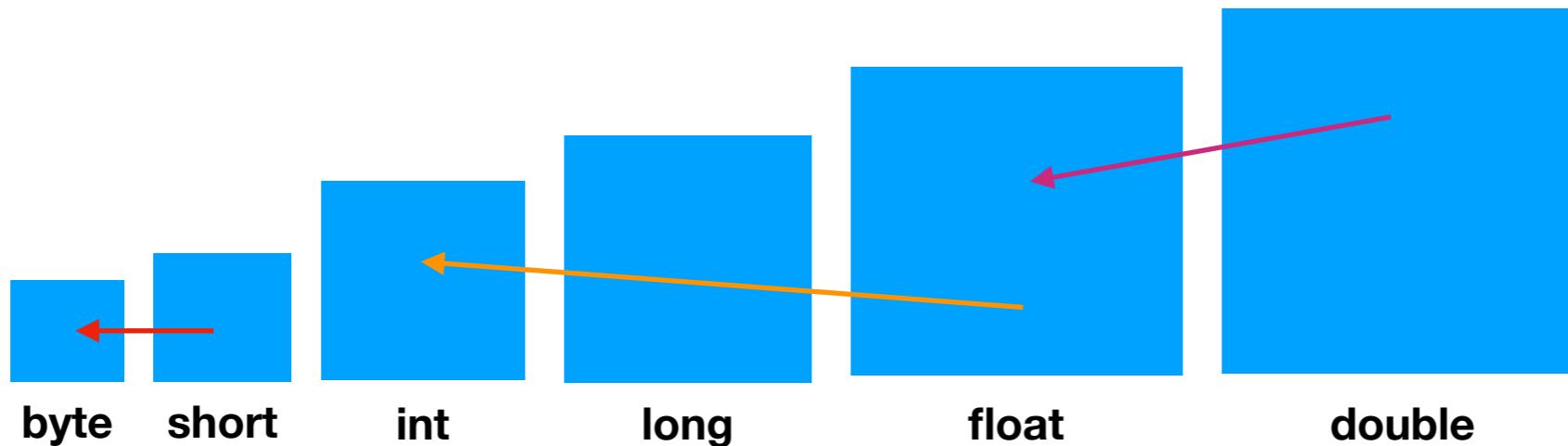


```
public static void main(String[] args){  
  
    byte num1 = 12;  
    short num2 = num1;  
    System.out.println(num2);  
  
    short num3 = 123;  
    int num4 = num3;  
    System.out.println(num4);  
  
    int num5 = 456789;  
    double num6 = num5;  
    System.out.println(num6);  
}
```



1) Explicit Narrowing Casting: If you assign a larger data type to a smaller data type then java **cannot convert** the data type automatically to the smaller one.

You have to convert manually



```
public static void main(String[] args){  
  
    short num1 = 12;  
    byte num2 = (byte) num1;  
    System.out.println(num2);  
  
    int num3 = 123;  
    short num4 = (short) num3;  
    System.out.println(num4);  
  
    double num5 = 456789;  
    int num6 = (int) num5;  
    System.out.println(num6);  
}
```



**11) Why is eclipse giving error in the following code?
Fix the bug in two different ways.**

```
public static void main(String[] args){  
  
    float num1 = 3.23;  
  
    double num2 = 3.23;  
  
}
```



12) What do you see on the console?

```
public static void main(String[] args){  
  
    /*  
     * byte is between -128 and 127  
     * If you type the following code, then what do you see on the console?  
     */  
  
    short num1 = 255;  
    byte num2 = (byte) num1;  
    System.out.println(num2);  
}
```



13) What do you see on the console?

```
public static void main(String[] args){  
  
    int num1 = 5/2;  
    System.out.println(num1);  
  
    float num2 = 5f/2f;  
    System.out.println(num2);  
  
    double num3 = 5d/2d;  
    System.out.println(num3);  
  
}
```



14) What do you see on the console?

```
public static void main(String[] args){  
  
    int num1 = 5/3;  
    System.out.println(num1);  
  
    float num2 = 5f/3f;  
    System.out.println(num2);  
  
    double num3 = 5d/3d;  
    System.out.println(num3);  
  
}
```



Wrapper Classes in Java

primitive

27

age

A

initial

Memory

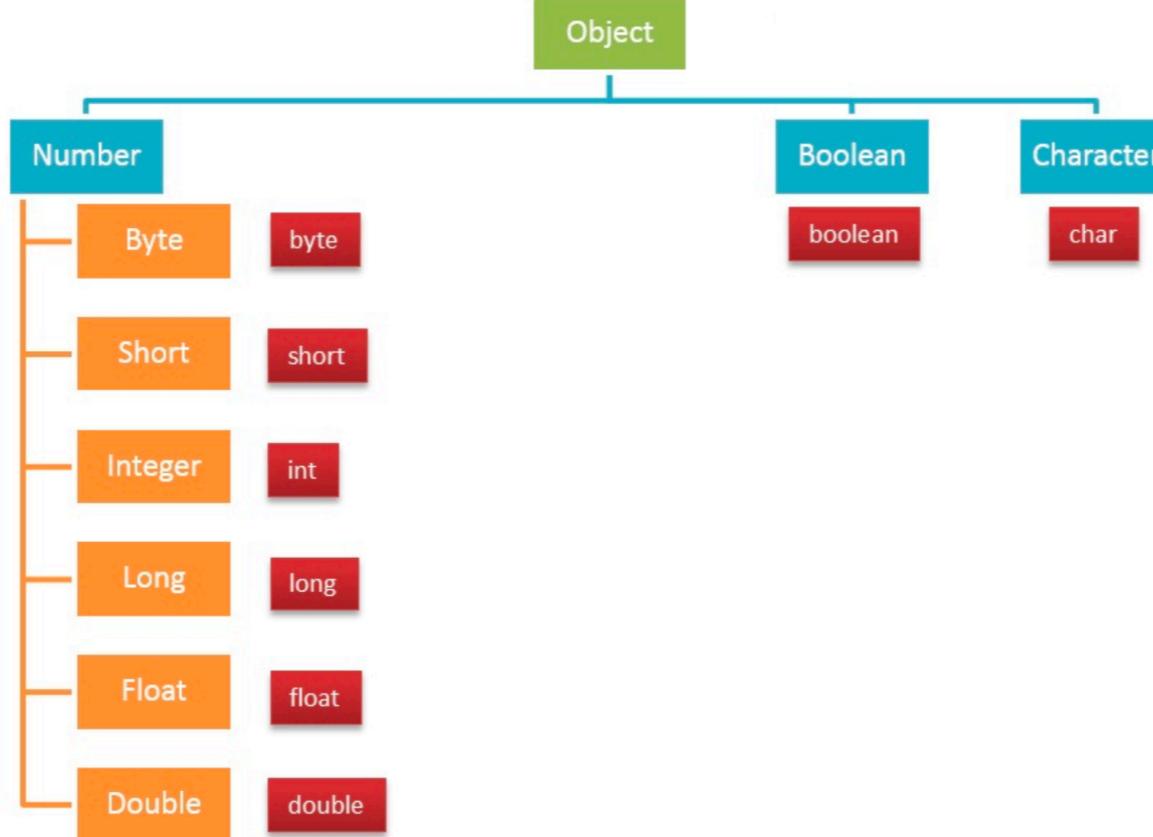
non-primitive

John

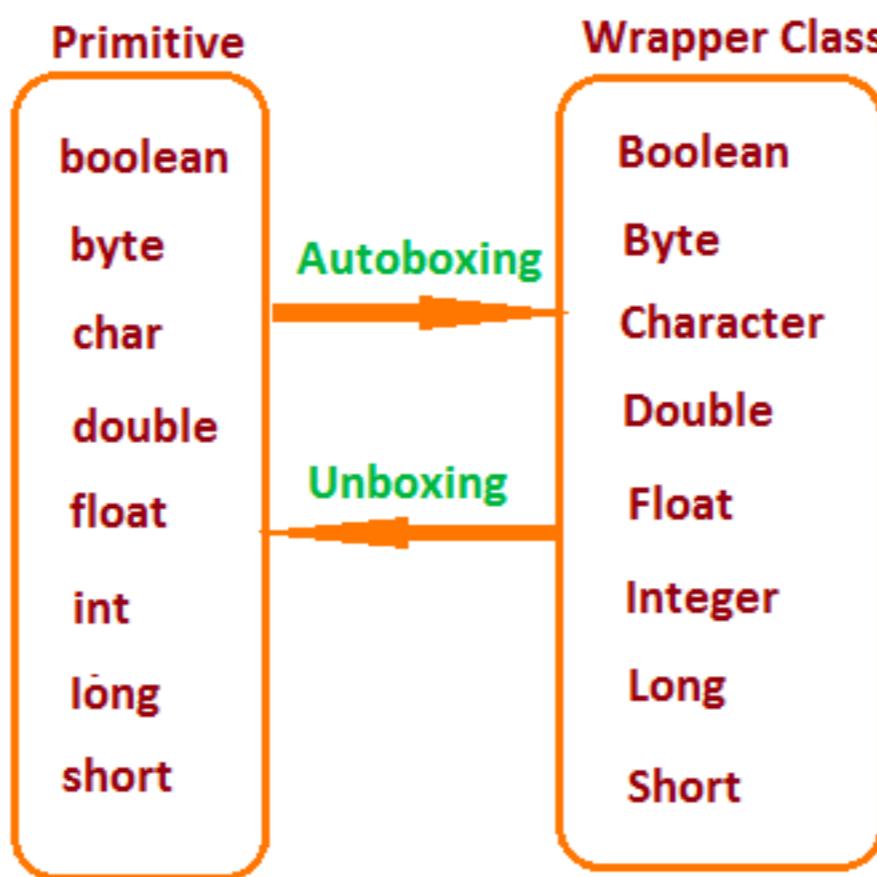
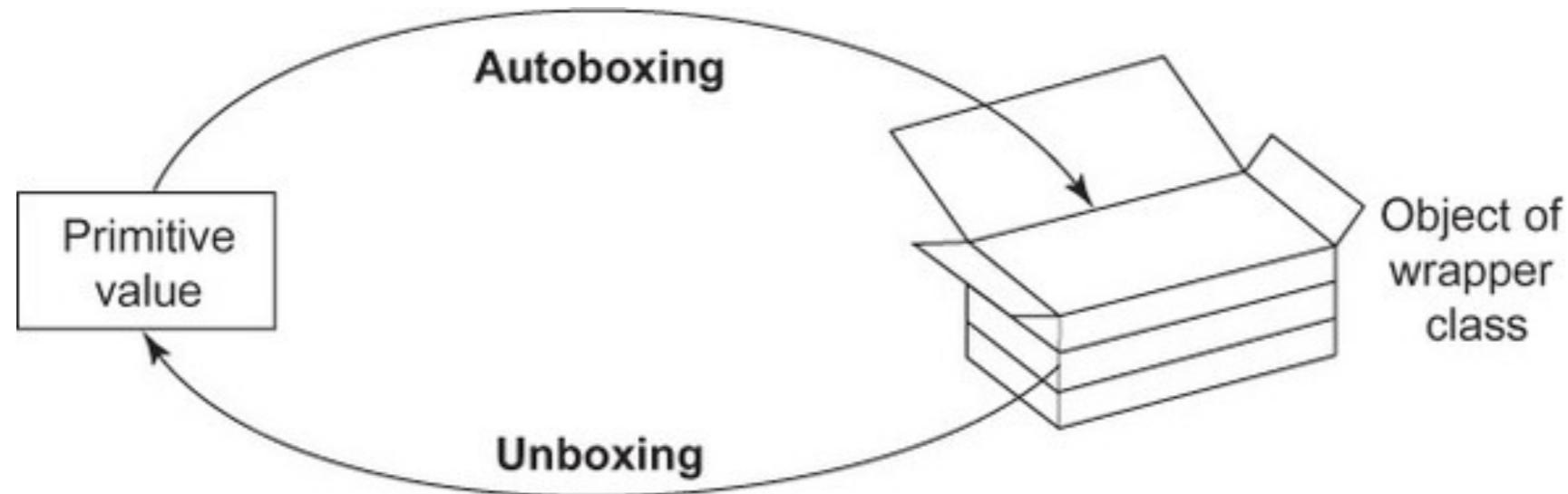
methods

name

Memory



Autoboxing & Unboxing in Java



How to find min and max values of primitive data types in eclipse ?

```
public static void main(String[] args){  
    int num1 = Integer.MIN_VALUE;  
    System.out.println(num1);  
    int num2 = Integer.MAX_VALUE;  
    System.out.println(num2);  
  
    int num3 = Byte.MIN_VALUE;  
    System.out.println(num3);  
    int num4 = Byte.MAX_VALUE;  
    System.out.println(num4);  
}
```



How to get data from user ?

- 1) **Scanner scan = new Scanner(System.in);**
↓
This is name if you want, you can change
- 2) **Give directions to user about what the user will do.**
System.out.println(“Enter two integers less than 100”)
- 3) **int num1 = scan.nextInt()**
int num2 = scan.nextInt()



How to get data from user

1. Scanner **scan** = new Scanner(**System.in**);
2. **System.out.println**("Enter an integer less than 30");
3. int **num1** = **scan.nextInt()**;
4. **System.out.println**(**num1**);

nextBoolean()	→ Reads a boolean value from the user
nextByte()	→ Reads a byte value from the user
nextDouble()	→ Reads a double value from the user
nextFloat()	→ Reads a float value from the user
nextInt()	→ Reads a int value from the user
nextLine()	→ Reads a String value from the user
nextLong()	→ Reads a long value from the user
nextShort()	→ Reads a short value from the user



- 1) Type a program which calculates the area and the perimeter of a square whose side length is entered by user.**
- Hint 1: Area of a square is length x length**
- Hint 2: Perimeter of a square is 4x length**



2) Type a program which calculates the cube of a number which is entered by user.

Hint 1: Cube of a number is $a \times a \times a$



3) Type a program which calculates the area and the perimeter of a rectangle whose length and width are entered by user.

Hint 1: Area of a rectangle is width x length

Hint 2: Perimeter of a rectangle is $2 \times (\text{width} + \text{length})$



- 4) Type a program which calculates the volume of a rectangular prism whose length, width, and height are entered by user.**
- Hint 1: Volume of a rectangular prism is width x length x height**



5) Type a program which calculates the area and the perimeter of a circle whose radius is entered by user. (Use float)

Hint 1: Take pi number as **3.14159**

Hint 2: Area of a circle is **3.14159 x radius x radius**

Hint 3: Perimeter of a circle is **2 x 3.14159 x radius**

Hint 4: To get float, use **nextFloat()**



6) Type a program which calculates the perimeter of a triangle whose Side lengths are entered by user. (Use byte)

Hint 1: Perimeter of a triangle is $a + b + c$

Hint 4: To get byte, use `nextByte()`



**For every question *create a class*, you can *use any name*,
then *create main method*....**

7) Type a program which converts the mile to kilometer. Mile value will be entered by user. (Use double)

Hint 1: km = mile x 1.6

Hint 2: To get double, use nextDouble()



**For every question *create a class*, you can *use any name*,
then *create main method*....**

8) Type a program which converts the hours to seconds. Hours value will be entered by user. (Use long)

Hint 1: second = hour x 60 x 60

Hint 2: To get long, use nextLong()



**For every question *create a class*, you can *use any name*,
then *create main method*....**

- 9) Type a program which asks user to enter his/her first name and last name,
then print it on the console.
Hint: To get String, use **nextLine()**



**For every question *create a class*, you can *use any name*,
then *create main method*....**

- 10) Type a program which asks user to enter his/her full name, and address
then print them on the console like the full name should be in the first line,
and the address will be in the second line.

Hint: To get String, use `nextLine()`



How to Increase Value of a Variable?

Increment

```
int numA = 2 ;  
numA = numA + 3;
```

or

```
numA += 3
```

numA = ?

```
int numB = 10 ;  
numB = numB * 7;
```

or

```
numB *= 7
```

numB = ?

```
int numC = 7 ;  
numC++ ;
```

```
int numD = 11 ;  
numD++ ;
```

numC = ?

numD = ?



How to Decrease Value of a Variable?

Decrement

```
int numA = 2 ;  
numA = numA - 3;
```

or

```
numA -= 3
```

numA = ?

```
int numB = 20 ;  
numB = numB / 5;
```

or

```
numB /= 5
```

numB = ?

```
int numD = 7 ;  
numD -- ;
```

numD = ?

```
int numE = 11 ;  
numE -- ;
```

numE = ?



Operators in Java

int a = 20 ; int b = 10 ; int c = a + b ; c = ? + —> Addition

int d = a - b ; d = ? - —> Subtraction

int e = a * b ; e = ? * —> Multiplication

int f = a / b ; f = ? / —> Division

Order of Operations

- 1) Do operations inside the parenthesis**
- 2) Do multiplications and divisions**
- 3) Do additions and subtractions**

Example:

$$38 / 2 - (4 + 3) * 2 = ?$$

Example:

$$8 + 2 * (14 - 6 / 2) - 12 = ?$$



15) Write a program to assign a value of 100.235 to a double variable and then convert it to int. Print it on the console.

16) Write a program to add an integer variable having value 5 and a double variable having value 6.2. Print the sum on the console.



17) Create an integer variable and increase it by 1, by using three different ways, then type every result on the console.



18) Create an integer variable and decrease it by 1, by using three different ways, then type every result on the console.



% → Modulus

int a = 20 ; int b = 8 ; int c = a % b ; c = ?

int a = 20 ; int b = 5 ; int d = a % b ; d = ?

```
public static void main(String[] args){  
  
    int num1 = 45;  
    int num2 = 13;  
    int remainder = num1 % num2;  
    System.out.println("Remainder: " + remainder);  
}
```



1) Type a program like;

Ask user to enter two integer values, the first will be greater than the second.
The remainder when you divide the first by the second will be the width,
and the sum of the two numbers will be the length of a rectangle.
Then calculate the area and the perimeter of the rectangle, and print them
on the console.

**2) Ask user to enter two integer values. Write a Java Program to swap
two numbers by using the third variable.**

**3) Ask user to enter two integer values. Write a Java Program to swap
two numbers without using the third variable.**



How to Join two Strings?

Concatenation

String str1 = “Learn” + “Java”;

str1 = ?

String str4 = “2” + “5” ;

str4 = ?

String str2 = “Learn” + “ ” + “Java”;

str2 = ?

String str5 = 2 + 3 + “4” ;

str5 = ?

String str3 = “Learn ” + “Java”;

str3 = ?

String str6 = “2” + (5 + 1) ;

str6 = ?



19) `int numA = 2;`
`int numB = 3;`
`String str1 = "Study"`
`String str2 = "Hard"`

Type a program to see the following outputs on the console by using “concatenation” operation.

- A) Study Hard B) 5 Study C) Hard23 D) Hard1

Note: Be careful about the spaces between the variables

20) `int numA = 2;`
`int numB = 3;`
`String str1 = "Study"`
`String str2 = "Hard"`

Type a program to see “**61Study-1**” as output on the console by using “concatenation” operation.

Note: Use just variable names, do not use any number



Operator Signs in Java

1) = → Assignment Operator in Java

```
int num1 = 12;  
boolean isOld = true;
```

2) == → Equal Sign or Comparison Operator in java

```
boolean isTrue = 5 + 2 == 7;  
boolean isFalse = 13 + 4 == 71;
```

3) != → Not Equal Sign in java

```
boolean isTrue = 5 + 2 != 7;  
boolean isFalse = 13 + 4 != 71;
```



4) “>” —> “*Greater than*” sign in java

boolean isTrue = 12 > 13

5) “<” —> “*Less than*” sign in java

boolean isTrue = 12 < 13

6) “>=” —> “*Greater than or equal to*” sign in java

boolean isTrue = 12 >= 13

boolean isTrue = 12 >= 12

7) “<=” —> “*Less than or equal to*” sign in java

boolean isTrue = 12 <= 13

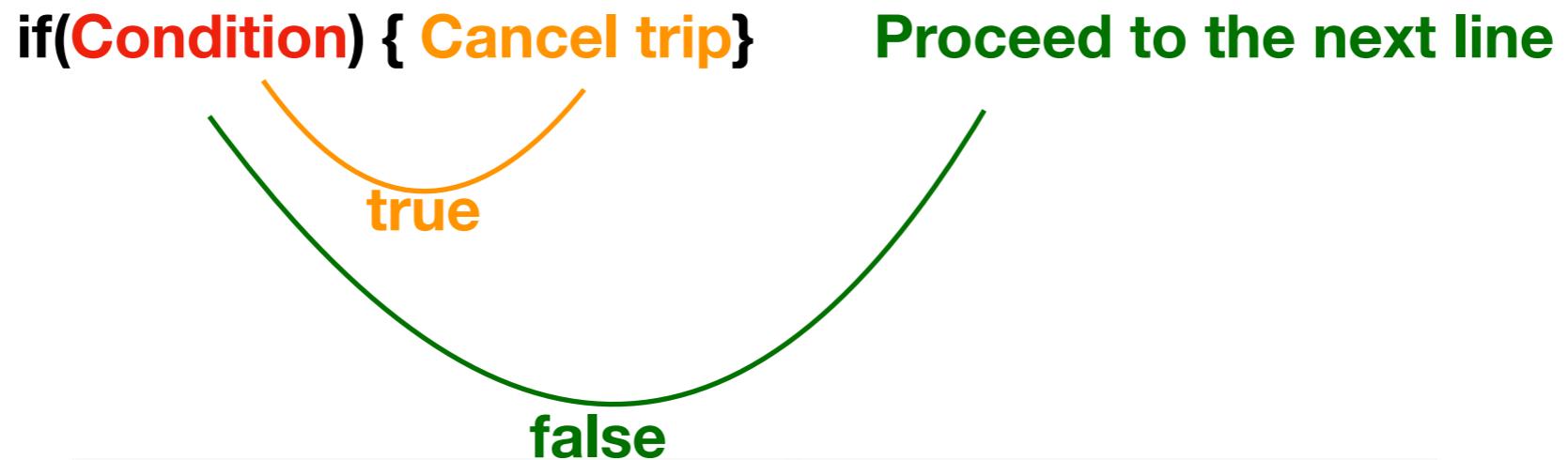
boolean isTrue = 12 <= 12



if Statement

If it rains, I will cancel the trip

If I study hard, I will get offer in a week



```
int num1 = 21;

if(num1 < 13){
    System.out.println("First if statement ran");
}

if(num1 > 19){
    System.out.println("Second if statement ran");
}

if(num1 % 4 == 5){
    System.out.println("Third if statement ran");
}

System.out.println("All if conditions checked");
```



if Statement Questions

- 1) Type java code, if an integer is even, output will be “The integer is even”.
If the integer is odd, output will be “The integer is odd”.**

- 2) Type java code by using if statement. When you enter the initial of the day of a week,
output should be all possible names of the days.
For example; if the initial is 'S' output should be “Saturday or Sunday”**

- 3) Type java code by using if statement. When you enter the name of the day of a week,
output will be “Weekday” or “Weekend day” according to the name of the day.**

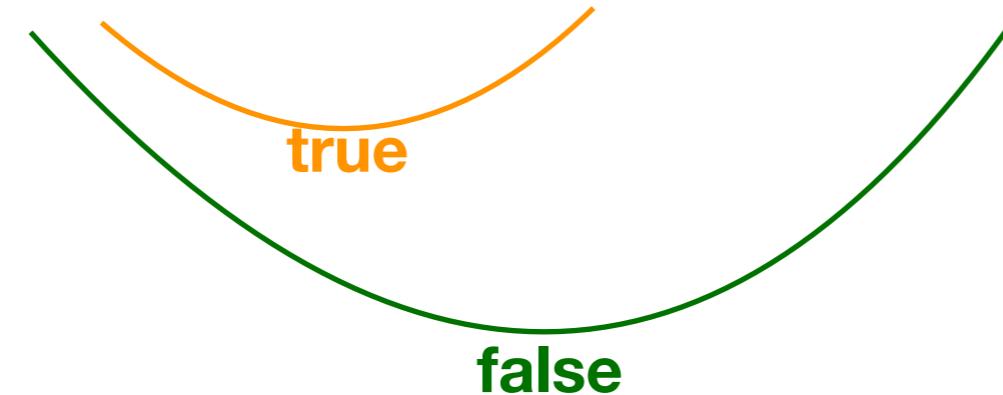
- 4) Ask user to enter the values of length and width of a rectangle, then check if it is square or not.**



if-else Statement

If it rains, I will go to mall; else I will have a picnic

if(Condition) { Go to mall } else {Have a picnic}



```
int number = 10;

if (number > 0) {
    System.out.println("Number is positive.");
}
else {
    System.out.println("Number is not positive.");
}

System.out.println("This statement is always executed.");
```



If-else Statement Questions

- 1)** Ask user to enter his/her age. If the age is between 18 and 65 then output will be “You should work”, else output will be “No need to work”

- 2)** Ask user to enter his/her age and gender. If the age is more than 65 and the gender is male then output will be “Hey man you retired!” else output will be “No need to work”

- 3)** Ask user to enter a character, then check whether the character is alphabet or not

- 4)** Ask user ta enter any name of the week, then get second ,fourth, and sixth letter of the day name and print them on the console, in the same line.
For example; if the user enters “Monday” output will be “ody”



- 5) Type java code by using if-else statement,
if the password is “JavaLearner”, output will be “The password is true”.
Otherwise, output will be “The password is false”.**
- 6) Type java code by using if-else statement,
Write a program to print absolute value of a number entered by user.
Absolute Value: If the number is positive or zero return the number itself
If the number is negative return the number after multiplying by -1**
- 7) Type java code by using if-else statement.
A shop will give discount of 10% if the cost of purchased quantity is more than 1000.
Ask user for quantity and unit price then judge and print total cost for user.
If the quantity is less than 1000 output will be “No discount.”**



Ask user ta enter a 4 digits integer, then print the sum of the first and the last digit of the number on console.

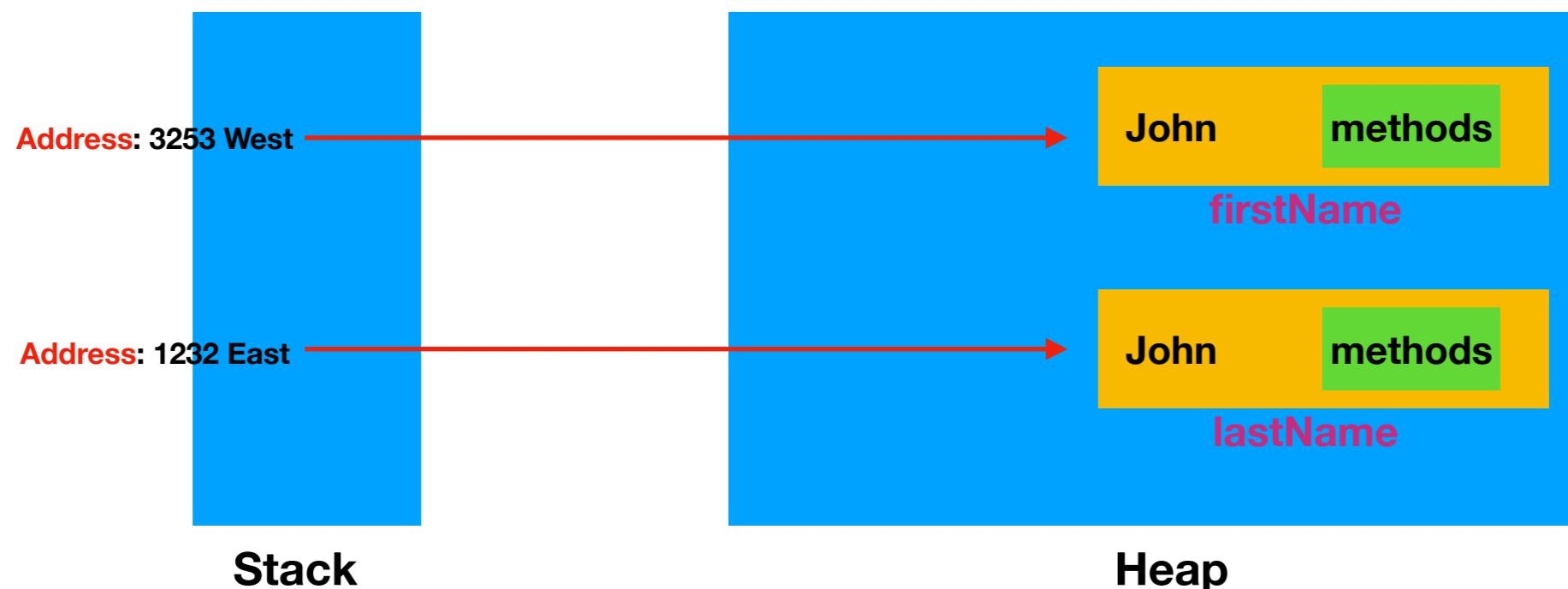
For example; if user enters 1234 you will add 1 and 4, then print on the console 5



Difference between “==” and “equals()”

`String firstName = "John";`

`String lastName = "John";`



Logical Operators

1) **&&** —> “And” operator

true && true = true
true && false = false
false && true = false
false && false = false



*If 0 is false and 1 is true,
“&&” operation is similar to multiplication of 0 and 1*

```
if ( 5 > 7 && 10 > 6 ) {  
    Apple  
} else {  
    Grape  
}
```

```
if ( 5 > 7 && 10 < 6 ) {  
    Apple  
} else {  
    Grape  
}
```

```
if ( 5 < 7 && 10 > 6 ) {  
    Apple  
} else {  
    Grape  
}
```

```
if ( 5 < 7 && 10 < 6 ) {  
    Apple  
} else {  
    Grape  
}
```



2) || -> “Or” operator

```
true || true = true  
true || false = true  
false || true = true  
false || false = false
```

If 0 is false and 1 is true,
|| operation is similar to addition of 0 and 1

```
if ( 5 > 7 || 10 > 6 ) {  
    Apple  
} else {  
    Grape  
}
```

```
if ( 5 < 7 || 10 > 6 ) {  
    Apple  
} else {  
    Grape  
}
```

```
if ( 5 > 7 || 10 < 6 ) {  
    Apple  
} else {  
    Grape  
}
```

```
if ( 5 < 7 || 10 < 6 ) {  
    Apple  
} else {  
    Grape  
}
```



Logical Operators

3) ! —> “**Not**” operator

! (true) = false

! (false) = true

```
if ( !(5 > 7) && 10 > 6 ) {  
    Apple  
} else {  
    Grape  
}
```

```
if ( 5 > 7 || !(10 > 6) ) {  
    Apple  
} else {  
    Grape  
}
```



Detail...

What is the difference between “&” and “&&” ?

40 < 30 & 5 > 2

Both conditions are checked every time; therefore, it is slow.

40 < 30 && 5 > 2

If the first condition is **false** then second condition is **not checked**; therefore, it is faster.

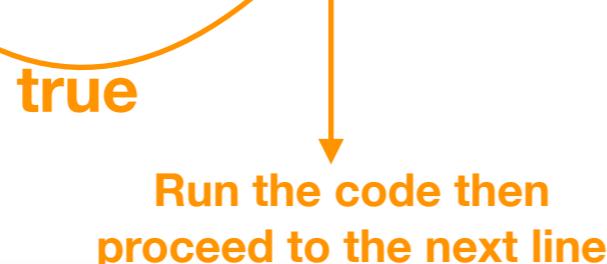
Use “**&&**” every time...



if - else if() Statement

If it rains, I will cancel the trip; else if it snows I will stay at home

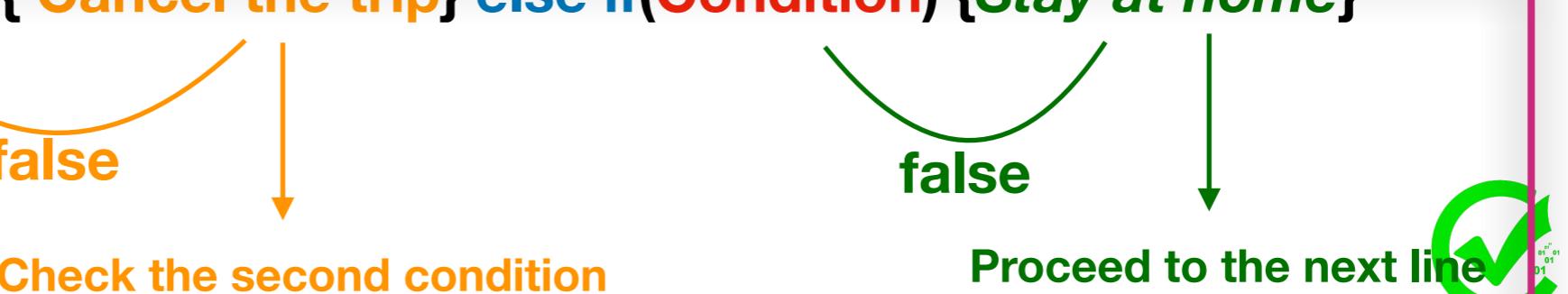
1) **if(Condition) { Cancel the trip} else if(Condition) {Stay at home}**



2) **if(Condition) { Cancel the trip} else if(Condition) {Stay at home}**



3) **if(Condition) { Cancel the trip} else if(Condition) {Stay at home}**



If-else if() Statement Questions

- 1)** Type java code by using if-else if() statement,
if both of the two integers are positive, output will be the sum of them.
If both of the two integers are negative, output will be the multiplication of them.
Otherwise; output will be “I cannot add or multiply different signed numbers”

- 2)** Type java code by using if-else if() statement,
If age is less than 13 output will be “Should not work”,
If age is greater than 65 output will be “Retired”,
Otherwise; output will be “Should work”

- 3)** Type java code by using if-else if() statement.
A school has following rules for grading system:
1. Below 50 - D **2.** 50 to 59 - C **3.** 60 to 80 - B. **4.** From 80 to 100 - A
Ask user to enter marks and print the corresponding grade.

- 4)** Ask user to enter year
Type java code by using if-else if() statement.
Write a program to check if a year is leap year or not.
if the year is divisible by 100 then it must be divisible by 400.
If a year is not divisible by 100 then it must be divisible by 4.

- 5)** Ask user to enter annual salary, if the salary is more than or equal
to \$80.000 output will be “I accept the offer”, if the salary is between
\$60.000 and \$80.000 out put will be “I negotiate to increase”,
otherwise output will be “I do not accept the offer.”



Nested if Statement

```
boolean male = false;
int age = 30;

if( male ){
    if( age < 20 ){
        System.out.println("Boy");
    }
    else{
        System.out.println("Man");
    }
}else{
    if( age < 20 ){
        System.out.println("Girl");
    }
    else{
        System.out.println("Woman");
    }
}
```



Nested if() Statement Questions

1) Type java code by using nested if statement,

If a number is even then check if it is divisible by 3 or not. If it is divisible by 3 the output will be “Perfect even number” otherwise, the output will be “Good even number.”

If the number is odd then check if it is divisible by 3 or not. If it is divisible by 3 the output will be “Perfect odd number” otherwise, the output will be “Good odd number.”

2) Type java code by using nested if() statement.

Write a program to check if a year is leap year or not.

If the year is divisible by 100 then it must be divisible by 400.

If a year is not divisible by 100 then it must be divisible by 4.

Type java code by using nested if() statement.

Ask user to enter a password

3)

If the initial of the password is uppercase then check if it is ‘A’ or not.

If it is ‘A’ the output will be “Valid Password”

otherwise the output will be “Invalid Password”

For example; Ali ==> Valid password - ali ==> Invalid - Mark ==> Invalid

If the initial of the password is lowercase then check if it is ‘z’ or not.

If it is ‘z’ the output will be “Valid Password”

otherwise the output will be “Invalid Password”

For example; zoe ==> Valid password - Zoe ==> Invalid - john ==> Invalid

Ternary Operator

if(Condition) { Code 1} else {Code 2}

true

false

==>

Condition ? Code 1 : Code 2

true

false

Example 1:

**int y = 10;
(y > 5) ? (2 * y) : (3 * y);**

OR

**int y = 10;
int x = (y > 5) ? (2 * y) : (3 * y);**

Example 2:

**int y = 7;
(y < 5) ? (2 * y) : (3 * y);**

OR

**int y = 7;
int x = (y < 5) ? (2 * y) : (3 * y);**



What do you see on the console?

Example 3:

```
int y = 112;  
System.out.println( (y > 5) ? (21) : ("Zebra") );
```

Example 4:

```
int y = 112;  
System.out.println((y < 91) ? 9 : "Horse");
```

Example 5:

```
int y = 11;  
int z = 11;  
int result = y<10 ? y++ : z++;  
System.out.println(result + "," + y + "," + z);
```



Ternary Operator Questions

- 1)** Type java code by using ternary and if-else, ask user to enter an integer, if the integer is even, the output will be “The integer is even”. If the integer is odd, the output will be “The integer is odd”.

- 2)** Type java code by using ternary and if-else. Ask user to enter two integers
Write a program to print the minimum one on the console.

- 3)** Type java code by using ternary.
Write a program to print absolute value of an integer entered by user.

- 4)** Type java code by using using ternary.
Take values of length and width of a rectangle from user and check if it is square or not.

- 5)** Ask user to enter a String. If the String has 2 characters, output will be
“It is valid for state abbreviations” Otherwise, output will be “It is not valid for state abbreviations”

- 6)** Ask user to enter an integer. If the number has 3 digits, output will be
“This number has 3 digits.” Otherwise, output will be “This number has no 3 digits.”
How can you decide the number of digits of an integer?

- 7)** Ask user ta enter a number. If the number is less than 10 and greater than or equal to 0, calculate its cube. Otherwise, calculate its square.
 $\text{Cube of } a = a \cdot a \cdot a$ $\text{Square of } a = a \cdot a$



Nested Ternary

Condition ? Code 1 : Code 2 ;

Condition ? Code 1 : Code 2

Condition ? Code 1 : Code 2

Example 1:

Find the output for y = 8 and y = 12 and y = 4 and y = 5

(y > 5) ? (y < 10 ? 2*y : 3*y) : (y > 10 ? 2+y : 3+y);

Question 1:

Type java code by using nested ternary.

Write a program to check if a year is leap year or not.

If the year is divisible by 100 then it must be divisible by 400.

If a year is not divisible by 100 then it must be divisible by 4.



Example 2:

Find the output for name = “Ali” and name = “Veli”

(name.length() > 3) ? (name.contains(“i”) ? “Veli” : “No name”) : (“Ali”);

Example 3:

Find the output for name = “Ali” and name = “Veli”

(name.equals(“Ali”)) ? (name.charAt(0)==‘a’ ? “Good” : “Bad”) : (“The worst”);



Switch Statement

Example 1:

```
String gender =“male”;  
  
switch(gender) {  
  
    case “female”:  
        System.out.println(“This is a girl”);  
        break;  
  
    case “male”:  
        System.out.println(“This is a boy”);  
        break;  
  
    default:  
        System.out.println(“Enter a valid gender”);  
  
}
```



Example 2: *int dayOfWeek = 5;*

```
switch(dayOfWeek) {  
    case 1:  
        System.out.println("Monday");  
        break;  
    case 2:  
        System.out.println("Tuesday");  
        break;  
    case 3:  
        System.out.println("Wednesday");  
        break;  
    case 4:  
        System.out.println("Thursday");  
        break;  
    case 5:  
        System.out.println("Friday");  
        break;  
    case 6:  
        System.out.println("Saturday");  
        break;  
    case 7:  
        System.out.println("Sunday");  
        break;  
    default:  
        System.out.println("Enter a valid day number");  
}
```



Note:

You can use the following data types in switch statements;

- int
- byte
- short
- char
- String

Note:

You cannot use long, double, float and boolean in switch statements

```
long num = 123;  
switch(num) {}
```

```
boolean isCold = true;  
switch(isCold) {}
```



Switch Statement Questions

- 1) If the user pressed 1, 2, 3 the program will print the number that is pressed; otherwise, program will print "Not allowed".**

- 2) Write a Java program user will choose answer among A, B, C, or D. If the answer is true, output will be “True.” If the answer is false, output will be “False”. Correct answer is ‘C’ for the multiple option question.**

- 3) Type java code by using switch statement.
A school has following rules for grading system:
1. For 50 - C 2. For 80 - B. 4. For 100 - A
Ask user to enter marks and print the corresponding grade.**

- 4) Ask user ta enter one of the ‘U’, ‘S’, and ‘A’.
*Then type a program by using “switch statement” to print “United” for ‘U’ “States” for ‘S’, and “America” for ‘A’***



Example 1:

String str = “Cat, caterpillar”

```
int idx1 = str.indexOf("c");
int idx2 = str.indexOf("cat");
int idx3 = str.indexOf("s");
int idx4 = str.indexOf("CAT");
int idx5 = str.indexOf("ter");
int idx6 = str.indexOf("pars");
```

```
int idx7 = str.indexOf("a" , 3 );
int idx8 = str.indexOf("at", 2);
int idx9 = str.indexOf("at", 8);
```



Example 2:

String str = “Cat, caterpillar”

```
int idx1 = str.lastIndexOf("a");
int idx2 = str.lastIndexOf("at");
int idx3 = str.lastIndexOf("s");
int idx4 = str.lastIndexOf("CAT");
int idx6 = str.lastIndexOf("pars");
```

```
int idx7 = str.lastIndexOf("a" , 3 );
int idx8 = str.lastIndexOf("a" , 5 );
int idx9 = str.lastIndexOf("at", 2);
int idx10 = str.lastIndexOf("at", 7);
```



Example 3:

String str = “Cat, caterpillar”

```
String str1 = str.substring(0);
String str2 = str.substring(3);
String str3 = str.substring(4);
String str4 = str.substring(15);
String str5 = str.substring(16);
```

```
String str6= str.substring(5,8);
String str7 = str.substring(3,4);
String str8 = str.substring(5,5);
String str8 = str.substring(8,5);
```



Example 4:

```
String str = “Cat, caterpillar”
String str1 = str.toUpperCase();
String str2 = str.toLowerCase();
String str3 = str.substring(4).toLowerCase();
String str4 = str.substring(3,8).toUpperCase();
String str5 = str.toUpperCase().toLowerCase();
```

```
String strA = “Cat ”
String str1 = strA.trim();
```

```
String strB = “ Cat”
String str2 = strB.trim();
```

```
String strC = “ Cat ”
String str3 = strC.trim();
```



Review Questions

- 1)** Ask user to enter an integer, if it is less than 10, calculate its square and print it on the console.
If it is greater than 10 multiply it by 2 and print it on the console. Otherwise keep the number same and print it on the console.

- 2)** Ask user to enter his kid's name, if the name contains "a" output will be "This name contains 'a'."
if the name contains "z" output will be "This name contains 'z'." Otherwise, output will be "This name contains neither 'a' nor 'z'."

- 3)** Ask user to enter a letter, if it is uppercase check it is before "F" or not in alphabetical order.
If it is before "F" in alphabetical order output will be " Big before F", otherwise output will be "Big after F." If it is lowercase check it is before "h" or not in alphabetical order.
If it is before "h" in alphabetical order output will be "Small before h", otherwise "Small after h"

- 4)** Ask user ta enter his/her first and last name. If the first name is longer
output will be "First name is longer." If the length of last name is equal
To the length of last name output will be "First name and last name
have same length." Otherwise, output will be "Last name is longer"



5) Ask user to enter a word which has 4 letters and output will be inverse of the word.

For example; if user enters “MARK” output will be “KRAM”

6) Ask user to enter a String and output will be the number of the characters in the String.

7) Ask user to enter password, if the password is okay for the following conditions output will be “Your password is created successfully.” If the password is not okay for any of the following conditions Output will be “Enter a new password according to the give conditions”

- 1. First letter must be uppercase**
- 2. Last letter must be lowercase**
- 3. Password must contain 6 characters**

8) Ask user to enter his/her first name, last name and Social Security Number.

Then type a program which makes

- a) initials of the first name and the last name in uppercase,
other characters will be in lowercase.**
- b) all characters except last 4 characters of the Social Security Number “ * ”.**

For example; Suleyman Alptekin *****5678



For Loop



```
for ( int i=4; i>1; i- - ) {  
    System.out.println( i );  
}
```



for-loop Questions

- 1) Write a program to print counting numbers from 10 to 57 on the console by using for-loop.**

- 2) Write a program to print even counting numbers from 100 to 43 on the console by using for-loop.**

- 3) Write a program to print odd counting numbers from 200 to 33 on the console by using for-loop.**

- 4) Write a program to add counting numbers from 23 to 57 by using for-loop.
Print the sum on the console**

- 5) Write a program to multiply counting numbers from 7 to 15 by using for-loop.
Print the multiplication on the console**

- 6) Write a program to print counting numbers which are less than 200 and divisible by 5
On the console by using for-loop.**



Wrapper Classes

primitive

27

age

A

initial

Memory

non-primitive

John

methods

name

Memory

Wrapper Classes

27 methods

age

A methods

initial

Memory

int —> Integer

double —> Double

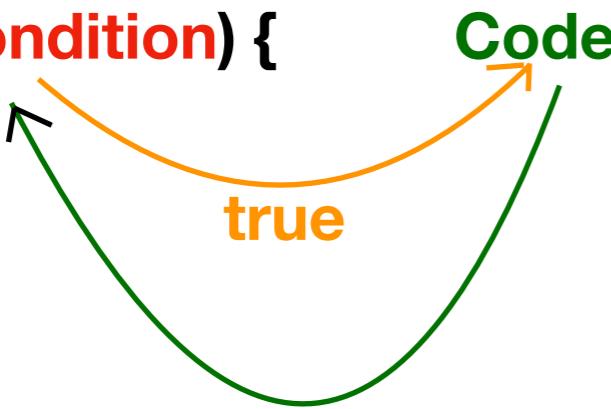
char —> Character



While Loop

While he was working, he listened to music.

while(condition) { **Code** **}**



After running the code check the condition again

while(condition) { **Code** **}**



Break the loop and proceed to the next line

```
int i = 0;  
while (i < 5) {  
    System.out.println(i);  
    i++;  
}
```



while loop Questions



4) Type java code by using while loop,
Write a program that prompts the user to input a positive integer.
It should then print factorial of that number.

5) Type java code by using while loop,
Write a program to count the factors of an integer which is entered by user.

6) Type java code by using while loop,
Write a program that prompts the user to input an integer.
It should then find sum of the digits of that number.



do-while Loop

do { Code } while(condition)

Run the code then check the condition

true

do { Code } while(condition)

Run the code then check the condition

false

Break the loop and proceed to the next line

```
int i = 0;
do {
    System.out.println(i);
    i++;
}
while (i < 5);
```

What is the difference between while and do-while loop?

do-while Loop Questions

- 1) Write a program to print numbers from 1 to 5 on the console by using do-while loop.**
- 2) Write a program to print numbers from 10 to 3 on the console by using do-while loop.**
- 3) Write a program to print numbers which are divisible by 5 between 1 and 100 on the console by using do-while loop.**
- 4) Write a program to print letters from c to m on the console by using do-while loop.**
- 5) Ask user to enter a number.
If the number is divisible by 10 then print "Won!" on the console
otherwise ask user to enter another number.
Use do-while loop.**
- 6) Ask user to enter a name.
If the name contains the letter 'a' then print "Won!" on the console
otherwise ask user to enter another name.
Use do-while loop.**
- 7) Ask user to enter a String
Print the characters whose indexes are odd on the console
For example; Germany ==> e m n**



Answer the following questions according to the given code.

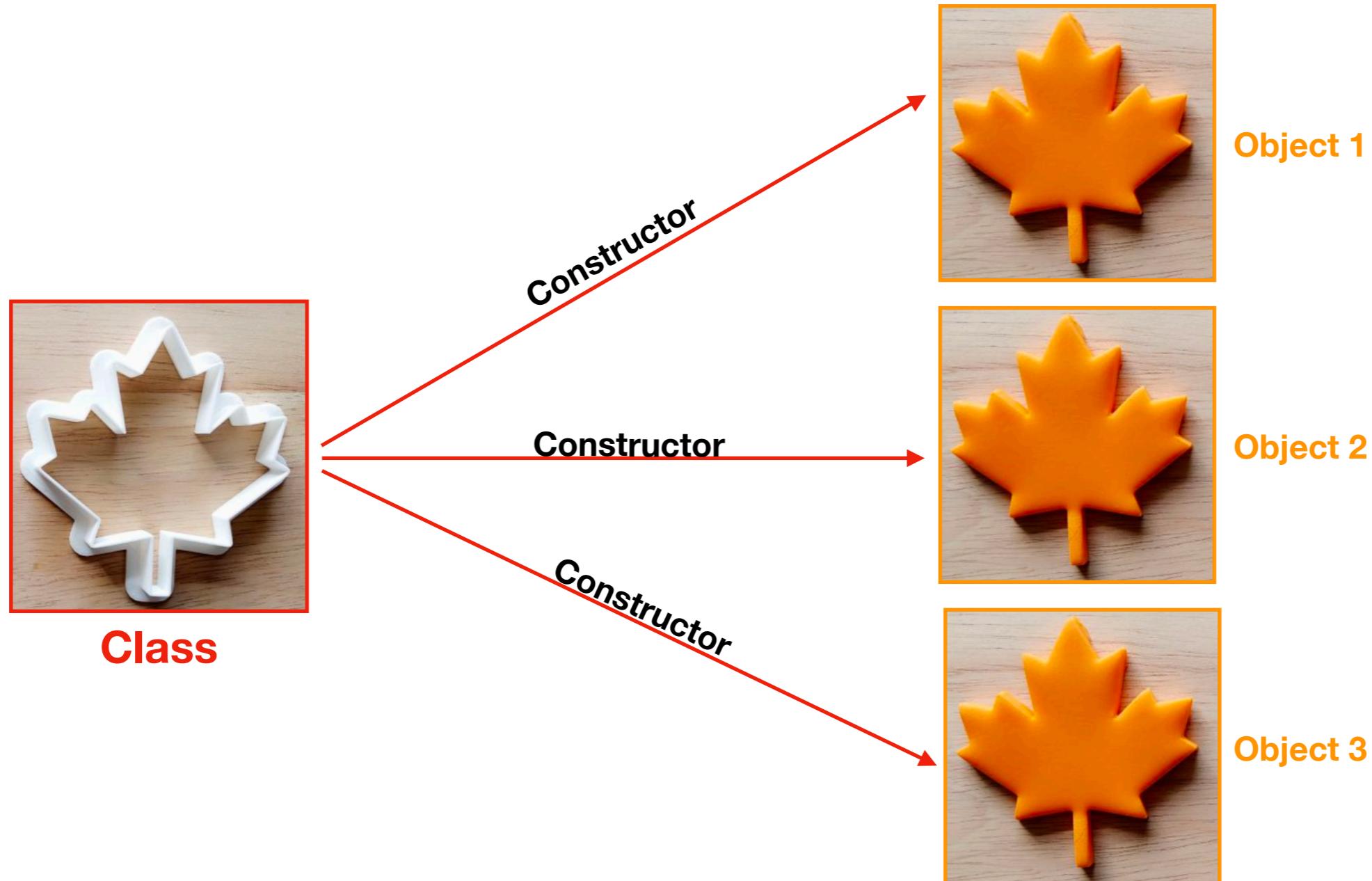
```
public class MyClass{  
    int num1;  
    String name = "Ali";  
  
    public static void main(String args){  
        add();  
        product(5);  
    }  
  
    public static add(){  
        num1++;  
        int num2 = 6;  
        char letter;  
        System.out.println("Do addition");  
    }  
  
    public product(int num3){  
        name = "Veli";  
        num2++;  
        System.out.println(num3 * num3);  
    }  
}
```

- 1) Which ones are instance variable?**
- 2) Which ones are local variables?**
- 3) What is the default value of num1?**
- 4) Which lines give red underline?**
- 5) How many lines give compile time error?**



How to create an object ?

Constructor is not a method, is not a variable, is a code block used to create objects from a class.



Constructor can be described as “Object Creator”



What is constructor?

```
public class MyClass {  
    MyClass() {}  
}
```

- 1) Constructor is inside the class
- 2) Constructor has the same name with the class
- 3) Constructor has no return type



Note 1: Constructor is not a method, because it has no return type.
Therefore, do not mention “constructor method”, mention just “constructor” while you talk about constructor

Note 2: When we create a class, Java creates a “default constructor” automatically for the class. So we can create objects by using every class without creating any constructor.

Note 3: Default constructors have no any parameters

```
MyClass() {  
}  
}
```

Note 4: When we create a constructor by ourselves, default constructor is cancelled by Java

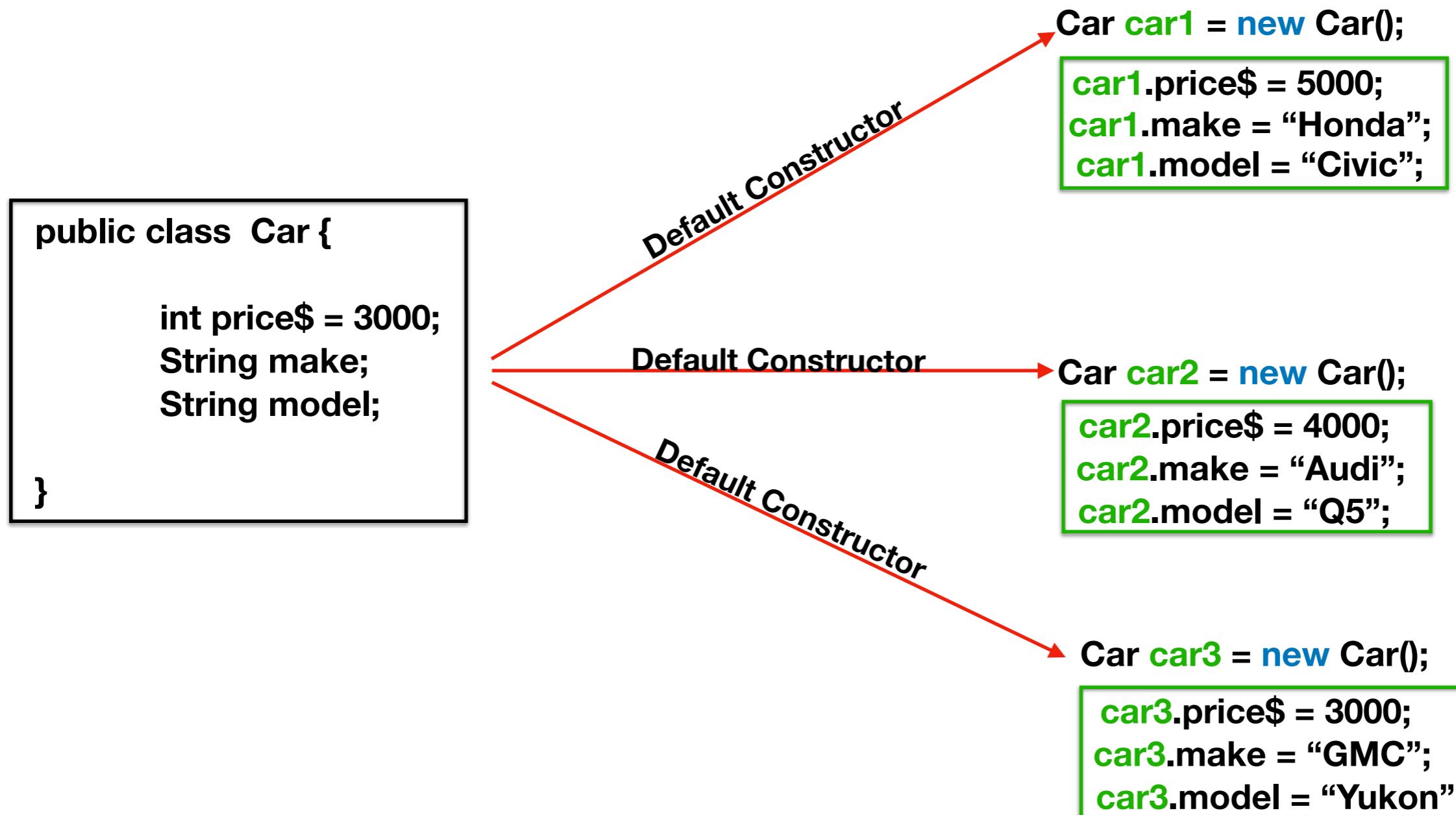
```
MyClass() {  
}  
}
```

```
MyClass(String str) {  
}  
}
```

```
MyClass(int i, String str) {  
}  
}
```



How does a constructor work ?



```
public class Car {  
  
    int price$;  
    String make;  
    String model;  
  
    public Car(){  
        this.price$;  
        this.make = "Honda";  
        this.model;  
    }  
}
```

No Parameters Constructor

No Parameters Constructor

No Parameters Constructor

Car car1 = new Car();

```
car1.price$ = 5000;  
car1.make = "Honda";  
car1.model = "Civic";
```

Car car2 = new Car();

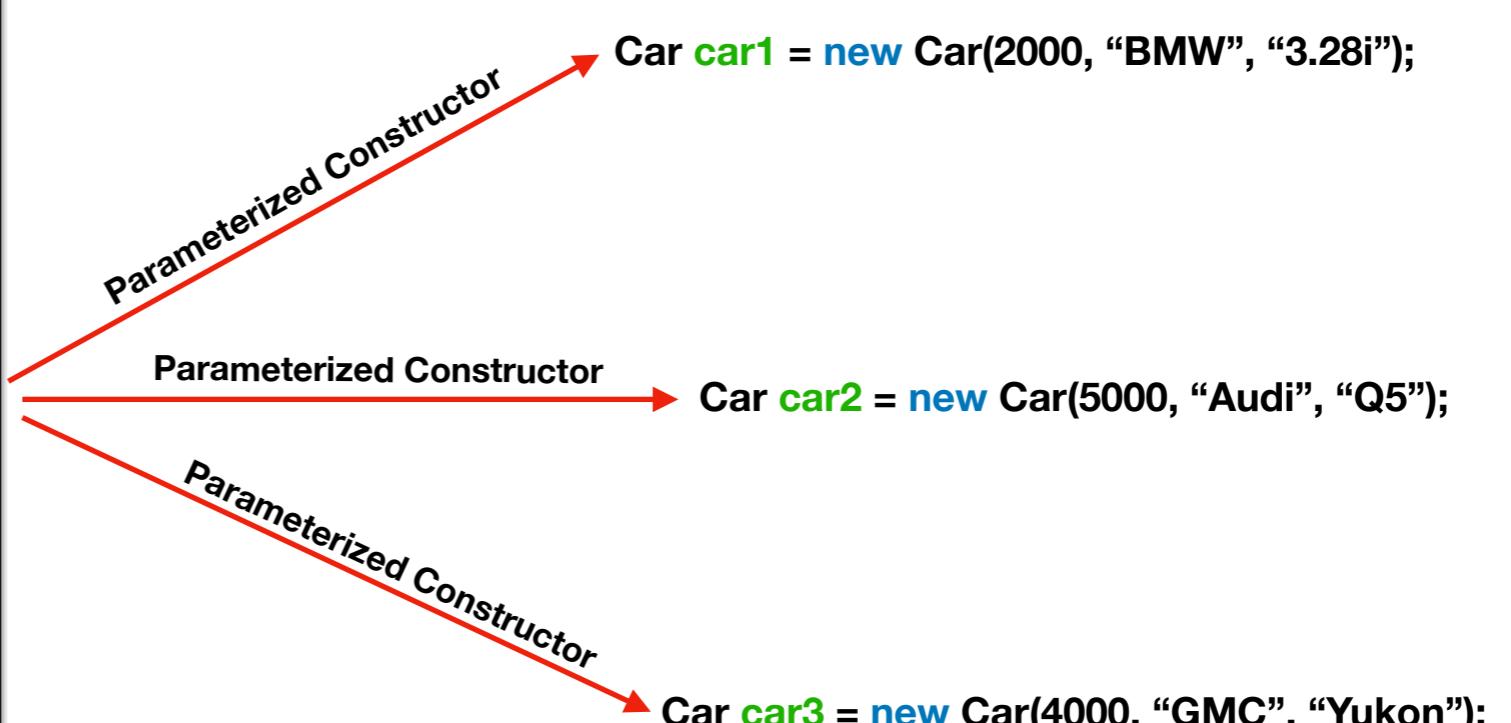
```
car2.price$ = 4000;  
car2.make = "Audi";  
car2.model = "Q5";
```

Car car3 = new Car();

```
car3.price$ = 3000;  
car3.make = "GMC";  
car3.model = "Yukon";
```

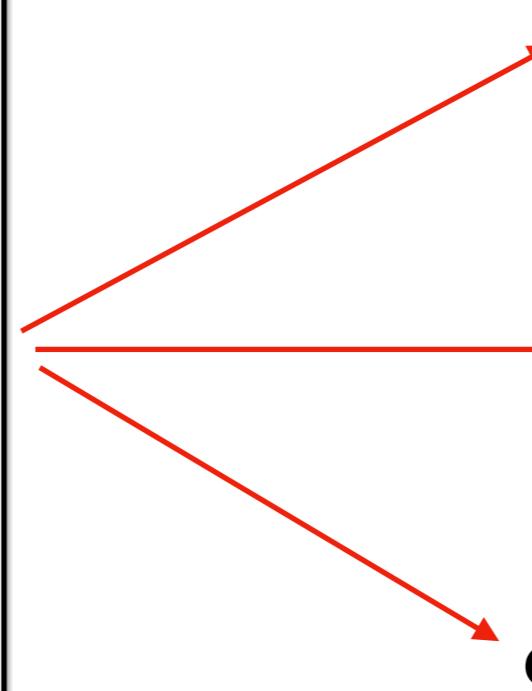


```
public class Car {  
  
    int price$;  
    String make;  
    String model;  
  
    public Car(int price$, String make, String model){  
        this.price$ = price$;  
        this.make = make;  
        this.model = model;  
    }  
}
```



What kind of car does every constructor create?

```
public class Car {  
  
    int price$;  
    String make;  
    String model;  
  
    public Car(int price$, String make, String model){  
        this.price$ = price$;  
        this.make = make;  
        this.model = model;  
    }  
}
```

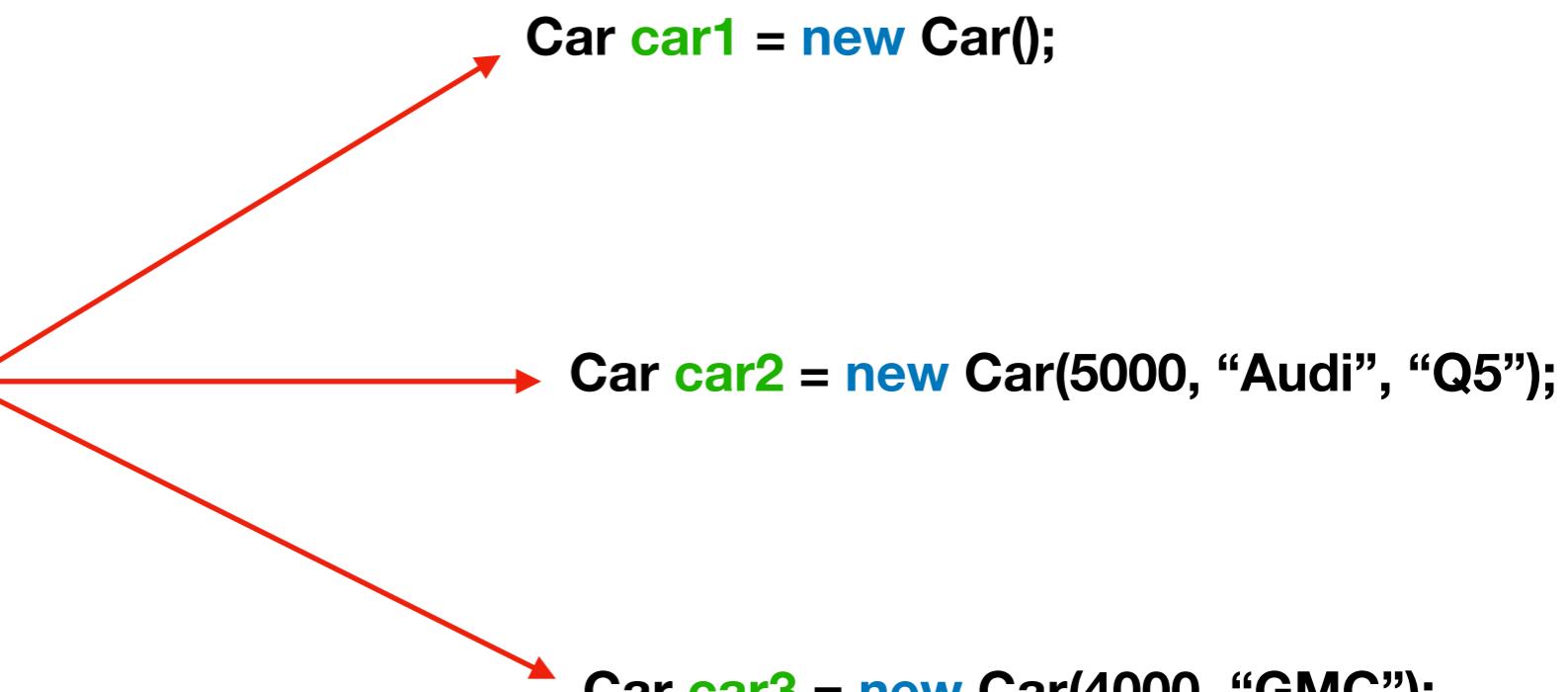


```
Car car1 = new Car();  
Car car2 = new Car(5000, "Audi", "Q5");  
Car car3 = new Car(4000, "GMC", "Yukon");
```



What kind of car does every constructor create?

```
public class Car {  
  
    int price$;  
    String make;  
    String model;  
  
    public Car(){  
        this.price$ = 2000;  
        this.make = "Honda";  
        this.model = "Accord";  
    }  
  
    public Car(int price$, String make, String model){  
        this.price$ = price$;  
        this.make = make;  
        this.model = model;  
    }  
}
```



What is the output?

```
public class Student
{
    String name = "Emily";
    int age = 20;

    Student(String name, int age)
    {
        this.name = name;
        this.age = 22;
    }
    public static void main(String[] args)
    {
        Student st = new Student("Oliver", 21);
        System.out.print(st.name);
        System.out.print(", " + st.age);
    }
}
```



What is the output?

```
public class Student {  
  
    String name;  
    int age;  
    String phone;  
  
    Student() {}  
    Student(String name, int age, String phone) {  
        this.phone = phone;  
        this.name = name;  
    }  
    public static void main(String[] args){  
        Student s1 = new Student();  
        Student s2 = new Student("John",25,"029-998877");  
        System.out.print(s2.name + ", " + s2.age + ", " + s2.phone)  
    }  
}
```



Question

```
public class MyClass{  
    int num1;  
    String name = "Ali";  
  
    MyClass(){  
        char letter = 'c';  
    }  
  
    MyClass(int num1){  
        this();  
        this.num1 = num1;  
    }  
  
    void MyClass(){  
        num1++;  
    }  
  
    increase(int num1){  
        name++;  
    }  
}
```

Fill in the blanks by “True” or “False”

- 1) **Turquoises** are instance variables.
- 2) **Orange** is an un-parameterized constructor.
- 3) **Pink** is a parameterized constructor.
- 4) **Green** is an un-parameterized constructor.
- 5) **Blue** is a parameterized constructor.
- 6) Variable “**letter**” is a local variable
- 7) Instance variables must be initialized
- 8) In the given code block, there is just one compile time error.
- 9) “**this**” keyword is related with instance variables
- 10) **this()** will call the green **MyClass()**



What is the output?

```
public class MyConstructor {  
    int x = 5;  
  
    MyConstructor() {  
        System.out.print("-x" + x);  
    }  
    MyConstructor(int x) {  
        this();  
        System.out.print("-x" + x);  
    }  
    public static void main(String[] args){  
        MyConstructor mc1 = new MyConstructor(4);  
        MyConstructor mc2 = new MyConstructor();  
    }  
}
```



What is the output?

```
public class MyClass {  
  
    int x = 3;  
    int y = 5;  
  
    MyClass() {  
        x += 1;  
        System.out.print("-x" + x);  
    }  
    MyClass(int i) {  
        this();  
        this.y = i;  
        x += y;  
        System.out.print("-x" + x);  
    }  
    MyClass(int i, int i2) {  
        this(3);  
        this.x -= 4;  
        System.out.print("-x" + x);  
    }  
    public static void main(String[] args){  
        MyClass mc1 = new MyClass(4,3);  
    }  
}
```



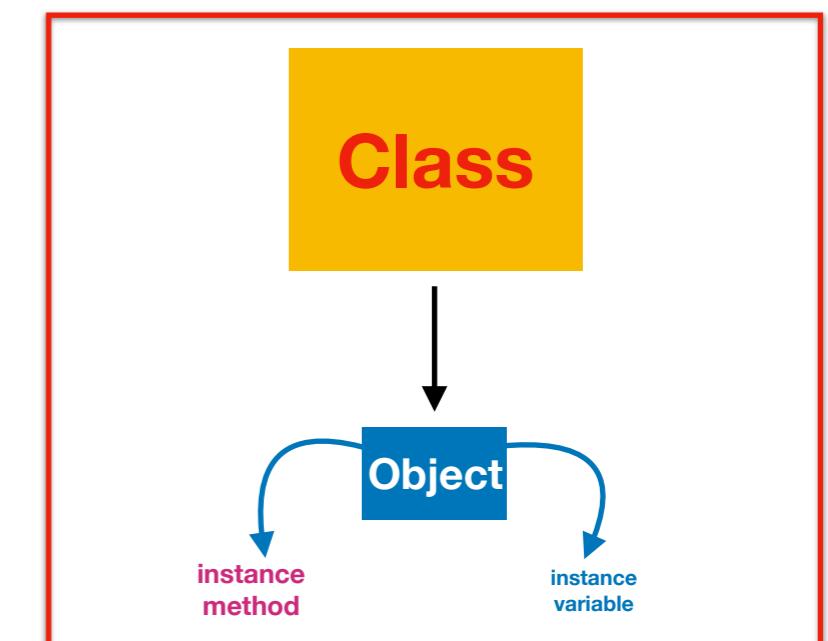
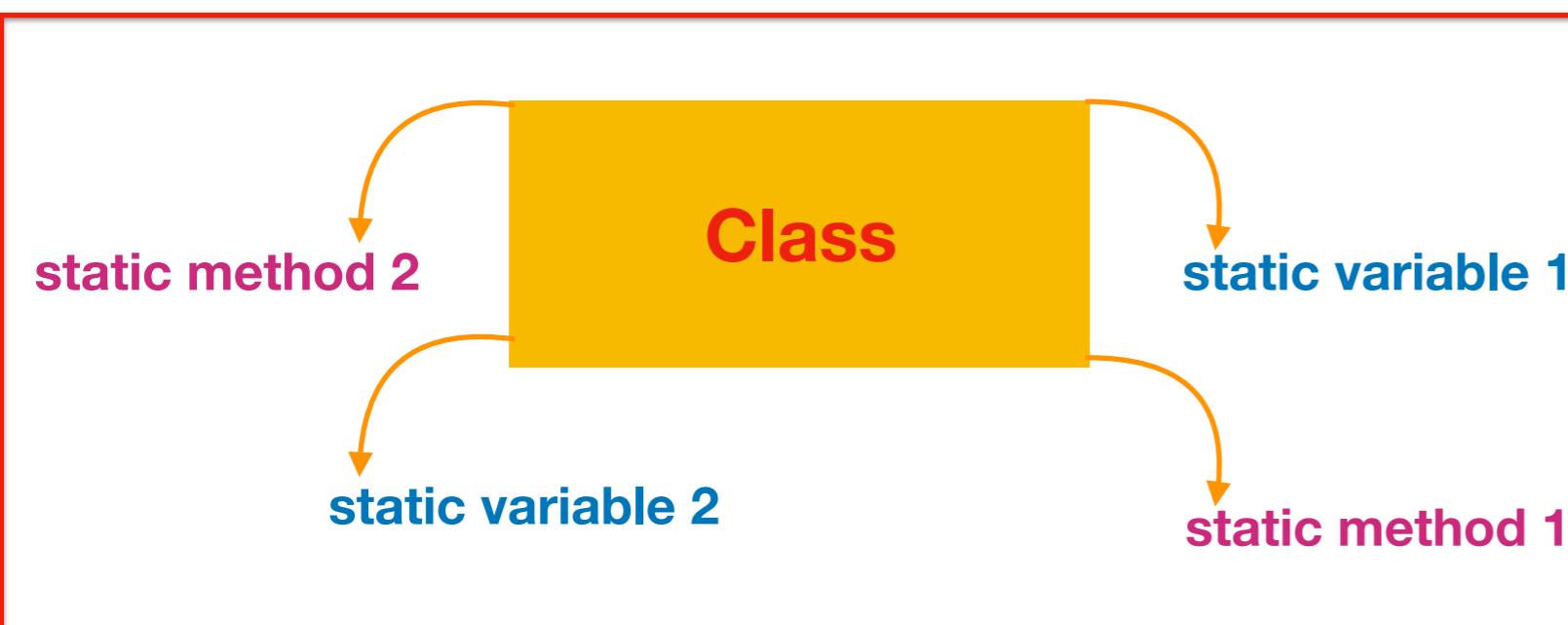
Purpose of Static Keyword in Java

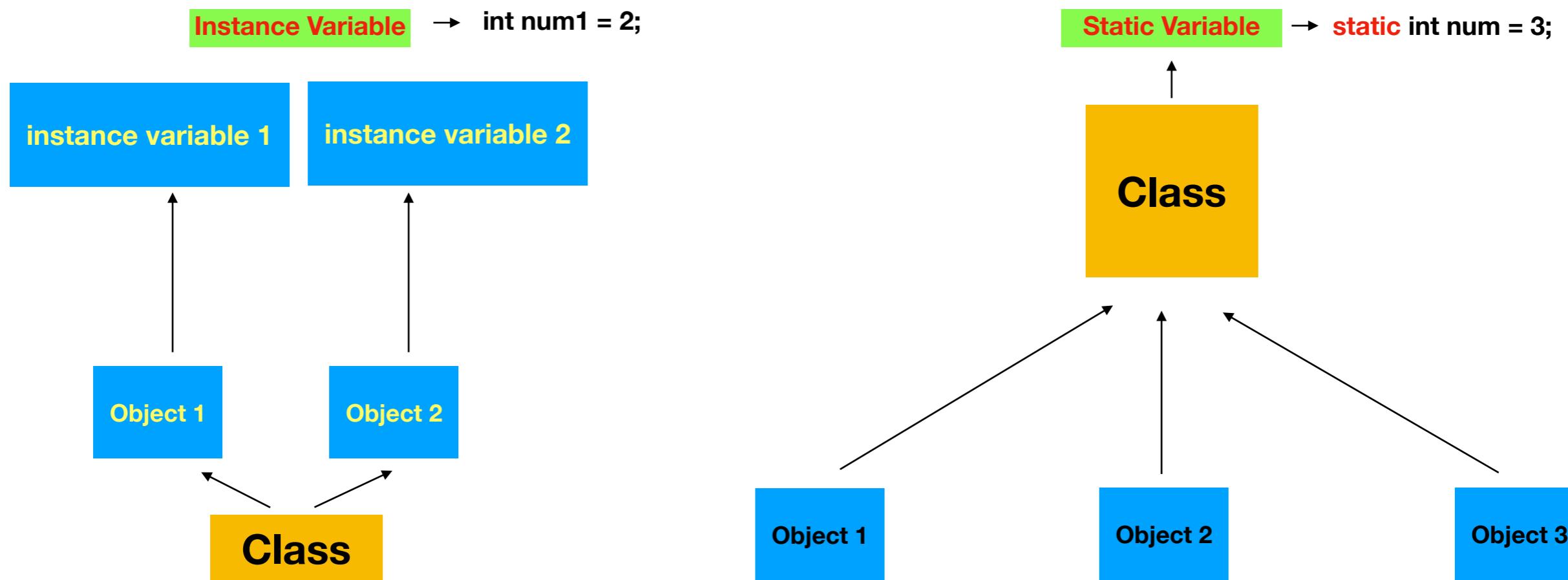
The static word can be used to attach a Variable or Method to a Class.

The Variable or Method that are marked static belongs to the Class rather than to any particular object

To call a static variable or method, no need to create an object

To call a non-static variable or method, you have to create an object





- 1)** When the class is loaded, in the memory static variables are initialized.
- 2)** Only a single copy of static variable is created and shared among all the instances of the class.
- 3)** Memory allocation for such variables only happens once.
- 4)** We can just use **Static Variables** in a static method
- 5)** Static Variable can be accessed by using class name without creating any object

Static Variable = Class Variable



What is the output?

```
class VariableDemo
{
    static int count=0;
    public void increment()
    {
        count++;
    }
    public static void main(String args[])
    {
        VariableDemo obj1=new VariableDemo();
        VariableDemo obj2=new VariableDemo();
        obj1.increment();
        obj2.increment();
        System.out.println("Obj1: count is "+obj1.count);
        System.out.println("Obj2: count is "+obj2.count);
    }
}
```



What is the output?

```
public class MyClass
{
    int x;
    static int y;

    MyClass(int i)
    {
        x += i;
        y += i;
    }
    public static void main(String[] args)
    {
        new MyClass(2);
        MyClass mc = new MyClass(3);
        System.out.print(mc.x + "," + mc.y);
    }
}
```



Static Method

```
public static void add(){  
    System.out.println(num1 + num2);  
}
```



By using **static keyword** before the return type,
we can create **static method**



1) Static Methods can access static variables(class variables) without using object of the class

```
class JavaExample{
    static int i = 10;
    static String s = "Beginnersbook";
    //This is a static method
    public static void main(String args[])
    {
        System.out.println("i:"+i);
        System.out.println("s:"+s);
    }
}
```



2) Static methods can be accessed directly in static and non-static methods.

```
class JavaExample{
    static int i = 100;
    static String s = "Beginnersbook";
    //Static method
    static void display()
    {
        System.out.println("i:"+i);
        System.out.println("i:"+s);
    }

    //non-static method
    void funcn()
    {
        //Static method called in non-static method
        display();
    }
    //static method
    public static void main(String args[])
    {
        JavaExample obj = new JavaExample();
        //You need to have object to call this non-static method
        obj.funcn();

        //Static method called in another static method
        display();
    }
}
```



What is the output?

```
public class Counter {  
    int count;  
    static int stCount;  
  
    public Counter() {  
        count ++ ;  
        stCount ++ ;  
    }  
    public int getCount(){  
        return count;  
    }  
    public static int getStCount(){  
        return stCount;  
    }  
}  
  
public class TestCounter {  
    public static void main(String[] args){  
        Counter cs1 = new Counter();  
        Counter cs2 = new Counter();  
        Counter cs3 = new Counter();  
        Counter cs4 = new Counter();  
        Counter cs5 = new Counter();  
        Counter cs6 = new Counter();  
        System.out.println("count is: " + cs6.getCount());  
        System.out.println("stCount is: " + cs6.getStCount())  
    }  
}
```



What is the output?

```
public class Counter {  
    int count;  
    static int stCount;  
  
    public Counter() {  
        count ++ ;  
        stCount ++ ;  
    }  
    public int getCount(){  
        return count;  
    }  
    public static int getStCount(){  
        return stCount;  
    }  
}  
  
public class TestCounter {  
    public static void main(String[] args){  
        Counter cs1 = new Counter();  
        Counter cs2 = new Counter();  
        Counter cs3 = new Counter();  
        Counter cs4 = new Counter();  
        Counter cs5 = new Counter();  
        Counter cs6 = new Counter();  
        System.out.println("count is: " + cs1.getCount());  
        System.out.println("stCount is: " + cs1.getStCount())  
    }  
}
```



What is the output?

```
public class StaticMember {  
  
    static int x;  
    int y;  
  
    StaticMember() {  
        x += 2;  
        y ++ ;  
    }  
    static int getSquare(){  
        return x * x;  
    }  
    public static void main(String[] args){  
        StaticMember sm1 = new StaticMember();  
        StaticMember sm2 = new StaticMember();  
        int z = sm1.getSquare();  
        System.out.print("-x" + z + "-y" + sm2.y);  
    }  
}
```



Instance Variable

- 1) Instance variables are declared in a , but outside ,
- 2) Instance variables are attached to an ; therefore, they are created when an is created and destroyed when the is destroyed.
- 3) Instance variables can be accessed by calling with the name.
- 4) Initialization of instance variable is not if it is not initialized it will have a default value
- 5) Every time, you will have the initialize value of an instance variable when you create a new object. True / False
- 6) If you created 6 objects by using a class which has two instance variables, it means you created 12 instance variables True / False

Static Variable

- 1) Static variables are declared in a , but outside ,
- 2) Static variables are attached to a ; therefore, they are created when a is created and destroyed when the is destroyed.
- 3) Static variables can be accessed by calling with the name.
- 4) Initialization of static variable is not if it is not initialized it will have a default value
- 5) Every time, you will have the initialize value of a static variable when you create a new class. True / False
- 6) If you created 6 objects by using a class which has two static variables, it means you created just 2 static variables. True / False



What is the output?

```
class Counter {  
    int count=0;  
  
    Counter(){  
        count++;  
        System.out.println(count);  
    }  
  
    public static void main(String args[]){  
  
        Counter c1=new Counter();  
        Counter c2=new Counter();  
        Counter c3=new Counter();  
    }  
}
```



What is the output?

```
class Student{  
    int number;  
    String name;  
    static String college ="ITS";  
  
    Student(int r, String n, String college){  
        this.number = r;  
        this.name = n;  
        this.college = college;  
    }  
  
    public static void main(String args[]){  
        Student s1 = new Student(111,"Karan", "MIT");  
        Student s2 = new Student(222,"Aryan", "Harvard");  
  
        System.out.println(s1.number);  
        System.out.println(s2.number);  
  
        System.out.println(s1.name);  
        System.out.println(s2.name);  
  
        System.out.println(s1.college);  
        System.out.println(s2.college);  
    }  
}
```



Static Block

- 1) Static block is used to initialize the static data member**
- 2) The static block is a block of statement inside a Java class that will be executed when a class is first loaded into the JVM**

```
class Test {  
    static int i;  
    int j;  
    static {  
        i = 10;  
        System.out.println("static block called ");  
    }  
    Test(){  
        System.out.println("Constructor called");  
    }  
}  
  
class Main {  
    public static void main(String args[]) {  
  
        // Although we have two objects, static block is executed only once.  
        Test t1 = new Test();  
        Test t2 = new Test();  
    }  
}
```

- 3) Static blocks are executed before constructors, and all methods and main method**
- 4) If you have more than one static block, the one above works first**



Arrays

Arrays are objects which **store multiple variables**.

Rule 1:

Arrays can contain just

- 1) Primitive data types
- 2) References of Objects

Primitive Data Types



References of Objects



Rule 2:

Arrays can store multiple variables of the **same data type**.

Rule 3:

We have to decide the capacity (**length**) of an array before starting to create

The number of data which you put inside an array can be less than length, but cannot be more than.



→ The max capacity (**length**) is 2



→ The max capacity (**length**) is 5

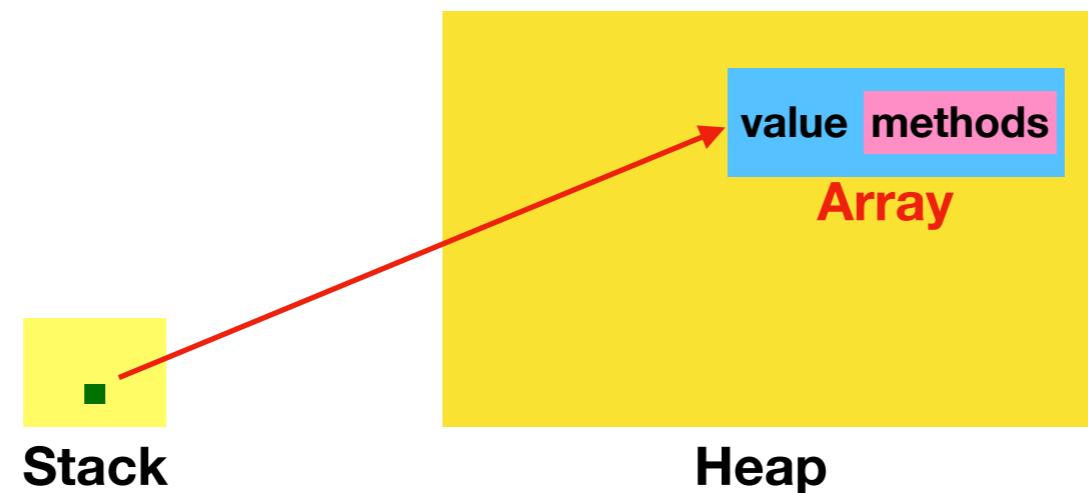


Point 1:

Arrays are objects; therefore, they are in Heap Memory

Point 2:

Arrays are objects; therefore, they have methods together with value



Point 3:

Arrays are objects; therefore, they are created during the runtime



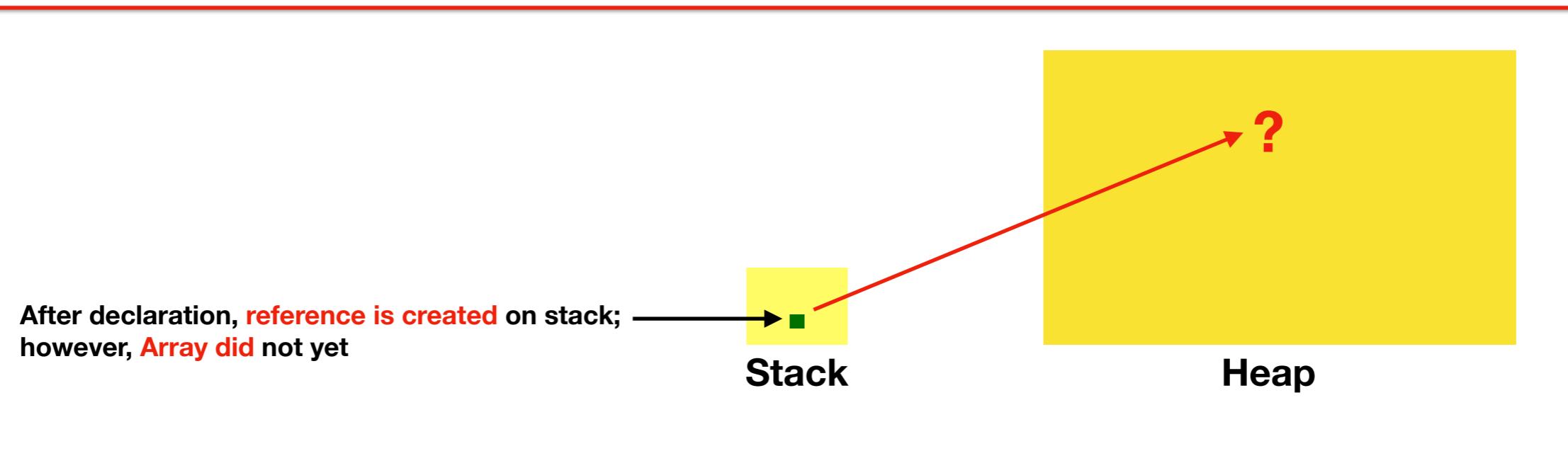
How to declare an array

Rule 4:

There are two ways to declare an Array :

- 1) `int myArray[];` //More common
- 2) `int[] myArray;`

Point 4:



How to construct an Array

Rule 5:

```
int myArray[ ] = new int[6];
```

- 1) The above code creates an Array whose **length is 6**.
- 2) We did not assign any value for the elements of the Array; therefore, values of all elements will be 0 as default.
If you print the Array on the console you will see {0, 0, 0, 0, 0, 0}

Point 5:

Note: If you do not declare the size during the construction, you will get compile time error



How to initialize an Array

Rule 6:

```
int myArray[] = new int[3];
```

```
myArray[0] = 9;  
myArray[1] = 10;  
myArray[2] = 11;
```

or

```
int myArray[] = {9, 10, 11};
```



How to access the elements of an Array

Rule 7:

```
int myArray[] = {9, 10, 11};
```

The array elements can be accessed with the help of the index.

myArray[0] refers to 9,

myArray[1] refers to 10,

myArray[2] refers to 10,

Point 6:

You can access up to myArray[n-1] where n is the size of the array.

Point 7:

Accessing the array with an index greater than or equal to the size of the array leads to “**ArraysIndexOutOfBoundsException**”.

How to get the size of an Array

Rule 8:

```
int myArray[] = {9, 10, 11};  
  
int size = myArray.length;
```

Point 8:

Be careful for the length of **Strings** we are using **length()**; however, for the size of the **Arrays** we are using **length**



How to print all the elements of an Array on the console

Rule 9:

```
int myArray[] = {9, 10, 11};  
int size = myArray.length;  
  
for(int i=0; i<size; i++) {  
    System.out.println(myArray[i]);  
}
```

Exercise:

Type a program like; given an array whose length is 3, return an array with the elements "rotated left"

For example; if the array is [1, 2, 3] output will be [2, 3, 1].



How to sort an Array

```
int[ ] numbers = { 6, 9, 1 };
```

```
Arrays.sort(numbers);
```

```
for (int i = 0; i < numbers.length; i++){
```

```
    System.out.print ( numbers[i] + " " );
```

```
}
```

→ Output is 1, 6, 9

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

```
    int arr[] = {2,1,7,6};
```

```
    Arrays.sort(arr);
```

```
    for(int i=0; i<arr.length; i++){
```

```
        System.out.print(arr[i] + " ");
```

```
}
```

→ ?

```
    String str[] = {"Ali", "Ahmet", "Kemal", "Can"};
```

```
    Arrays.sort(str);
```

```
    for(int i=0; i<str.length; i++){
```

```
        System.out.print(str[i] + " ");
```

```
}
```

→ ?

How to use `binarySearch()` in Arrays Class

Java also provides a convenient way to search by using `binarySearch()` in Arrays Class.

Note: To use `binarySearch()`, array must be sorted

```
int[ ] numbers = { 2, 4, 6, 8 };
```

```
System.out.println( Arrays.binarySearch(numbers, 2)); =====> 0
```

```
System.out.println( Arrays.binarySearch(numbers, 4)); =====> 1
```

Note: If an element does not exist in array

- 1) Find the index of the element if it exists
- 2) Increase the index by 1
- 3) Change the sign to negative

```
System.out.println( Arrays.binarySearch(numbers, 1)); =====> -1
```

```
System.out.println( Arrays.binarySearch(numbers, 3)); =====> -2
```

```
System.out.println( Arrays.binarySearch(numbers, 9)); =====> -5
```



What is the output?

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

```
    int arr[] = {2,1,7,6};
```

```
    Arrays.sort(arr);
```

```
    System.out.println(Arrays.binarySearch(arr, key: 2));
```

```
    System.out.println(Arrays.binarySearch(arr, key: 7));
```

```
    System.out.println(Arrays.binarySearch(arr, key: 3));
```

```
    System.out.println(Arrays.binarySearch(arr, key: 9));
```

```
    String str[] = {"A", "C", "B", "D"};
```

```
    Arrays.sort(str);
```

```
    System.out.println(Arrays.binarySearch(str, key: "A"));
```

```
    System.out.println(Arrays.binarySearch(str, key: "C"));
```

```
    System.out.println(Arrays.binarySearch(str, key: "E"));
```

```
    System.out.println(Arrays.binarySearch(str, key: "G"));
```

```
}
```

→ ?

→ ?

→ ?

→ ?

→ ?

→ ?

→ ?

→ ?



How to verify two arrays are equal to each other

Note 1: equals() returns boolean

Note 2: equals() compares both values and indexes

```
public static void main(String args[]){  
  
    int arr1[] = {2,1,7,6};  
    int arr2[] = {2,1,7,6};  
    System.out.println(Arrays.equals(arr1,arr2)); → ?  
  
    int arr3[] = {3,2,8,7,11};  
    int arr4[] = {7,8,3,11,2};  
    System.out.println(Arrays.equals(arr3,arr4)); → ?  
  
    int arr5[] = {4,5,9,8,10};  
    int arr6[] = {8,9,4,10,5};  
    Arrays.sort(arr5);  
    Arrays.sort(arr6);  
    System.out.println(Arrays.equals(arr5,arr6)); → ?  
}
```



How to print arrays on the console by using `toString()`

```
public static void main(String aargs[]) {  
    int arr[] = {1, 2, 3, 4, 5};  
    System.out.println(Arrays.toString(arr)); → [1, 2, 3, 4, 5]  
}
```

One more String method `split()`

`String str = “What about Java, did you like it”`

`String arr[] = str.split(“,”);`

`arr[] = { “What about Java”, “ did you like it” }`



Multi Dimensional Arrays

```
public static void main(String aargs[]) {  
    int arr[][] = new int[3][2];  
  
    arr[0][0] = 1;  
    arr[0][1] = 2;  
  
    arr[1][0] = 3;  
    arr[1][1] = 4;  
  
    arr[2][0] = 5;  
    arr[2][1] = 6;  
  
    System.out.println(Arrays.toString(arr[0]));  
    System.out.println(Arrays.toString(arr[1]));  
    System.out.println(Arrays.toString(arr[2]));  
  
    System.out.println(arr[0][1]);  
    System.out.println(arr[1][0]);  
    System.out.println(arr[2][1]);  
}
```

Creation Multi Dimensional Array

Initializing Multi Dimensional Array

Printing arrays inside the array

Printing the elements of inner arrays



Shorter way

```
public static void main(String aargs[]) {
```

```
    int arr[][] ={{1,2}, {3,4}, {5,6}};
```

Creating and initializing Multi Dimensional Array

```
    System.out.println(Arrays.toString(arr[0]));
    System.out.println(Arrays.toString(arr[1]));
    System.out.println(Arrays.toString(arr[2]));
```

Printing arrays inside the array

```
    System.out.println(arr[0][1]);
    System.out.println(arr[1][0]);
    System.out.println(arr[2][1]);
```

Printing the elements of inner arrays

```
}
```



Multi Dimensional Arrays Questions

- 1) Find the sum of all elements in the multi dimensional array { {1,2,3}, {4,5,6} }**

- 2) Find the product of the last elements in the array elements of the given multi dimensional array { {1,2,3}, {4,5}, {6} }**

- 3) Find the sum of the elements whose indexes are same in the given two multi dimensional arrays
arr1 = { {1,2}, {3,4,5}, {6} } and arr2 = { {7,8,9}, {10,11}, {12} }**

- 4) Find the sum of the elements in the array elements of the given multi dimensional array { {1,2,3}, {4,5}, {6,7} } and return an array.
For example; for { {1,2,3}, {4,5}, {6,7} } output will be {6, 9, 13}**

- 5) Ask user to enter long two sentences. Then type a program to count all “words” in the sentences.
For example; if user enters “Java is easy, if you study. Nothing is easy, if you do not study” output will be 14.
Hint: Use split()**

- 6) Write a Java program to remove a specific element from an array.**



ArrayList

What is ArrayList?

ArrayList is a resizable Array.

Why do we need ArrayList?

While we create an array, we have to declare the size of an Array. The size of an Array cannot be modified.

If you want to add elements to an Array or remove elements from an Array, you have to create a new Array.

However, we can add elements to an ArrayList and remove elements from ArrayList whenever we want. No need to create a new ArrayList

```
int arr[ ] = new int[ 5 ];
```

```
ArrayList<String> list1 = new ArrayList<String>();  
ArrayList<String> list2 = new ArrayList<>();  
List<String> list3 = new ArrayList<>();
```

```
ArrayList<String> list4 = new List<>(); // DOES NOT COMPILE
```

Note: ArrayList implements `toString()` so you can easily see the contents just by printing it.

```
List<String> list3 = new ArrayList<>();  
System.out.println(list3);
```



add()

The add() methods insert a new value in the ArrayList.

Example:

```
ArrayList<String> birds = new ArrayList<>(); // List<String> birds = new ArrayList<>(); is also fine
```

add() without index

```
birds.add("hawk"); // [hawk]
birds.add("test"); // [hawk, test]
```

add() with index

```
birds.add(1, "robin"); // [hawk, robin, test]
birds.add(0, "blue jay"); // [blue jay, hawk, robin, test]
birds.add(1, "cardinal"); // [blue jay, cardinal, hawk, robin, test]
```

```
System.out.println(birds); // [blue jay, cardinal, hawk, robin, test]]
```



size()

size() method looks at how many of the elements are in the ArrayList

```
List<String> birds = new ArrayList<>();  
System.out.println(birds.size()); // 0  
  
birds.add("hawk");  
birds.add("test");  
System.out.println(birds.size()); // 2
```

isEmpty()

isEmpty() method looks at the ArrayList whether if it is empty or not

```
List<String> birds = new ArrayList<>();  
System.out.println(birds.isEmpty()); // true  
  
birds.add("hawk");  
birds.add("test");  
System.out.println(birds.isEmpty()); // false
```



remove()

The **remove()** methods remove a value in the **ArrayList**.

remove() without index

The **remove()** without index methods remove the first matching value in the **ArrayList**

```
List<String> birds = new ArrayList<>();
```

```
birds.add("hawk"); // [hawk]  
birds.add("test"); // [hawk, test]
```

The remove() without index returns boolean

```
System.out.println(birds.remove("cardinal")); // prints false because "cardinal" is not in the list
```

```
System.out.println(birds.remove("hawk")); // prints true because "hawk" is in the list
```

The remove() with index returns the removed element

```
System.out.println(birds.remove(0)); // prints test
```

```
System.out.println(birds); // []
```

```
System.out.println(birds.remove(100)); // throws an IndexOutOfBoundsException
```



set()

The **set()** method changes one of the elements of the **ArrayList** without changing the size.

```
List<String> birds = new ArrayList<>();
```

```
birds.add("hawk"); // [hawk]
```

```
birds.add("test"); // [hawk, test]
```

```
System.out.println(birds.size()); // 2
```

```
birds.set(0, "robin"); // [robin, test]
```

```
System.out.println(birds.size()); // 2
```

Note: **set()** method cannot be used like **add()** method

```
birds.set(2, "robin"); // IndexOutOfBoundsException
```



contains()

The **contains()** method checks whether a certain value is in the ArrayList.

```
List<String> birds = new ArrayList<>();  
birds.add("hawk", "test"); // [hawk, test]
```

```
System.out.println(birds.contains("hawk")); // true  
System.out.println(birds.contains("test")); // true  
System.out.println(birds.contains("robin")); // false
```



sort()

The **sort()** method sorts the elements.

```
List<String> birds = new ArrayList<>();  
birds.add("B");  
birds.add("A");  
birds.add("C");  
System.out.println(birds); // [B, A, C]  
System.out.println(Collections.sort("birds")); // [A, B, C]
```

```
List<Integer> nums = new ArrayList<>();  
nums.add(3);  
nums.add(2);  
nums.add(7);
```

```
System.out.println(nums); // [3, 2, 7]  
System.out.println(Collections.sort("nums")); // [2, 3, 7]
```



equals()

By using **equals()** method we can compare two lists to see if they contain the same elements in the same order.

```
List<String> one = new ArrayList<>();
```

```
List<String> two = new ArrayList<>();
```

```
System.out.println(one.equals(two)); // true
```

```
one.add("a"); // [a]
```

```
System.out.println(one.equals(two)); // false
```

```
two.add("a"); // [a]
```

```
System.out.println(one.equals(two)); // true
```

```
one.add("b"); // [a,b]
```

```
two.add(0, "b"); // [b,a]
```

```
System.out.println(one.equals(two)); // false
```



clear()

The **clear()** method provides an easy way to discard all elements of the **ArrayList**.

Returns nothing, its return type is **void**

```
List<String> birds = new ArrayList<>();  
birds.add("hawk"); // [hawk]  
birds.add("hawk"); // [hawk, hawk]
```

```
System.out.println(birds.isEmpty()); // false  
System.out.println(birds.size()); // 2
```

```
birds.clear(); // []
```

```
System.out.println(birds.isEmpty()); // true  
System.out.println(birds.size()); // 0
```



List Question 01:

1) Create a String list whose elements are A, C, E, and F. Print it on the console.

2) By using add() with index method, add B into the 1st index.

List elements should be like A, B, C, E, and F. Print it on the console

3) By using set() method, convert E to D.

List elements should be like A, B, C, D, and F. Print it on the console

4) By using remove() method, remove F from the list.

List elements should be like A, B, C, D. Print it on the console

5) Find the size of the list and print the size on the console.

List Question 02:

Find the sum of the elements in the array { {1,2,3}, {4,5}, {6,7} } and return an array.

For example; for { {1,2,3}, {4,5}, {6,7} } output will be {6, 9, 13}



How to convert an ArrayList to an Array

```
List<String> list = new ArrayList<>();  
list.add("hawk");  
list.add("robin");  
System.out.println(list); // [hawk, robin]
```

```
String arr[ ] = list.toArray(new String[0]);  
System.out.println(arr.length); // 2  
System.out.println(Arrays.toString(arr)); // [hawk, robin]
```



How to convert an Array to an ArrayList

```
String[] arr = { "hawk", "robin" };           // [hawk, robin]
List<String> list = Arrays.asList(arr);       // converts the array to fixed size list
System.out.println(list.size());               // 2
System.out.println(list);                     // [hawk, robin]
```

Note: If you update the elements through Array methods or List methods, both array elements and list elements will be affected. Because they point to the same data store.

```
list.set(1, "test");                         // [hawk, test]
arr[0] = "new";                             // [new, test]
System.out.println(Arrays.toString(arr));     // [hawk, robin]
System.out.println(list);                   // [hawk, robin]
```

Note:

`list.remove(1); // throws UnsupportedOperationException because we are not allowed to change the size of the list. It is fixed size list`



For-each Loop / Enhanced For Loop

Advantages:

It makes the code more readable.

It eliminates the possibility of programming errors.

```
public static void main(String args[ ]){  
  
    int arr[ ]={12,13,14,44};  
  
    for( int i : arr ) {  
        System.out.print(i + " ");  
    }  
  
}
```



For-each Loop / Enhanced For Loop

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

```
    ArrayList<String> list=new ArrayList<String>();  
    list.add("Ali");  
    list.add("Veli");  
    list.add("Can");
```

```
    for( String s : list ) {  
        System.out.print(s + " ");  
    }
```

```
}
```



For-each Question 1:

Create an integer array find the sum of all elements by using for-each loop and print the sum on the console.

For-each Question 2:

Create a list find the sum of all elements by using for-each loop and print the sum on the console.

For-each Question 3:

Write a Java program to find the common elements between two arrays (string values).



Java uses “Pass by Value”

Animation to understand Pass by Value and Pass by Reference

<https://www.youtube.com/watch?v=wWh4U4Np05w>

```
public static void main(String aargs[]) {  
    double price = 100;  
    discountForVeteran(price);  
    discountForSeniors(price);  
    System.out.println(price);  
}  
  
public static void discountForVeteran(double price){  
    price = price*0.80;  
    System.out.println(price);  
}  
  
public static void discountForSeniors(double price){  
    price = price*0.90;  
    System.out.println(price);  
}
```



Pass by Value and Pass by Reference Example

```
public class ReturningValues {
```

```
    public static void main(String[ ] args) {
        int number = 1;                                // 1
        String letters = "abc";                         // abc
        number(number);                               // 1
        letters = letters(letters);                   // abcd
```

```
        System.out.println(number + letters); // 1abcd
    }
```

```
    public static int number(int number) {
        number++;
        return number;
    }

}
```



Local Date

```
System.out.println( LocalDate.now() ); // 2020 - 03 - 10
```

Local Time

```
System.out.println( LocalTime.now() ); // 12:45:18.401
```

Note: This time displays hours, minutes, seconds, and nanoseconds.

Local Date and Time

```
System.out.println( LocalDateTime.now() ); // 2020 - 03 - 10 T 12:45:18.401
```

Note: Java uses T to separate the date and time when converting LocalDateTime to a String..



Manipulating Dates

```
LocalDate date = LocalDate.now();  
System.out.println(date); // 2020 - 03 - 10
```

1) date = date.plusDays(1);

```
System.out.println(date); // 2020 - 03 - 11
```

OR

date = date.minusDays(1);

```
System.out.println(date); // 2020 - 03 - 09
```

2) date = date.plusMonths(2);

```
System.out.println(date); // 2020 - 05 - 10
```

OR

date = date.minusMonths(2);

```
System.out.println(date); // 2020 - 03 - 10
```

3) date = date.plusYears(3);

```
System.out.println(date); // 2023 - 05 - 10
```

OR

date = date.minusYears(3);

```
System.out.println(date); // 2017 - 03 - 10
```



Manipulating Times

```
LocalTime time = LocalTime.now();
```

```
System.out.println(time); // 12:45:18.401
```

```
time = time.plusMinutes(1);  
System.out.println(time); // 12:46:18.401
```

OR

```
time = time.plusHours(2);  
System.out.println(time); // 14:45:18.401
```

OR

```
time = time.plusSeconds(3);  
System.out.println(time); // 12:45:21.401
```

OR

```
time = time.plusNanos(5);  
System.out.println(time); // 12:45:18.406
```

OR

```
time = time.minusMinutes(1);  
System.out.println(time); // 12:44:18.401
```

```
time = time.minusHours(2);  
System.out.println(time); // 10:45:18.401
```

```
time = time.minusSeconds(3);  
System.out.println(time); // 12:45:15.401
```

```
time = time.minusNanos(5);  
System.out.println(time); // 12:45:18.396
```



Formatting Date

```
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd-MMM-yyyy");
```

```
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("MM dd yyyy");
```

```
LocalDate date = LocalDate.now();
```

```
dtf.format(date);
```

Formatting Time

```
DateTimeFormatter dtf = DateTimeFormatter.ofPattern("hh:mm");
```

```
LocalTime time = LocalTime.now();
```

```
dtf.format(time);
```



Working with Varargs

Note 1: Vararg Parameter (*variable argument*) as if it is an array.

Note 2: A vararg parameter must be the last element in a method's parameter list.

```
public void walk2( int start, int... nums ){} // COMPILE
```

Note 3: Being last implies you are only allowed to have one vararg parameter per method.

```
public void walk( int... nums, int... nums ){} // DOES NOT COMPILE
```



Example for Varargs:

```
public static void main (String[ ] args) {  
  
    walk(1);           // [ ] -----> Output is 0  
    walk(2, 2);        // [2] -----> Output is 1  
    walk(1, 2, 3);     // [2, 3] -----> Output is 2  
  
    walk(1, new int[] {4, 5}); // [4, 5] -----> Output is 2  
}
```

```
public static void walk(int start, int... nums) {  
  
    System.out.println(nums.length);  
}
```



Questions

1)

What is the result of the following code? (Choose all that apply)

```
13: String a = "";
14: a += 2;
15: a += 'c';
16: a += false;
17: if ( a == "2cfalse") System.out.println("==");
18: if ( a.equals("2cfalse")) System.out.println("equals");
```

- A. Compile error on line 14.
- B. Compile error on line 15.
- C. Compile error on line 16.
- D. Compile error on another line.
- E. ==
- F. equals
- G. An exception is thrown.



2)

What is the result of the following statements?

```
6: List<String> list = new ArrayList<String>();  
7: list.add("one");  
8: list.add("two");  
9: list.add(7);  
10: for(String s : list) System.out.print(s);
```

- A. onetwo
- B. onetwo7
- C. onetwo followed by an exception
- D. Compiler error on line 9.
- E. Compiler error on line 10.



3) What is the result of the following statements?

```
3: ArrayList<Integer> values = new ArrayList<>();  
4: values.add(4);  
5: values.add(5);  
6: values.set(1, 6);  
7: values.remove(0);  
8: for (Integer v : values) System.out.print(v);
```

- A.** 4
- B.** 5
- C.** 6
- D.** 46
- E.** 45
- F.** An exception is thrown.
- G.** The code does not compile.



4)

What is the result of the following?

```
int[] random = { 6, -4, 12, 0, -10 };  
int x = 12;  
int y = Arrays.binarySearch(random, x);  
System.out.println(y);
```

- A. 2
- B. 4
- C. 6
- D. The result is undefined.
- E. An exception is thrown.
- F. The code does not compile.



Access Modifiers

An access modifier **restricts** the **access** of a **class**, **constructor**, **data member**, and **method** in another class

We have 4 different access modifiers in Java

1) private

2) default →

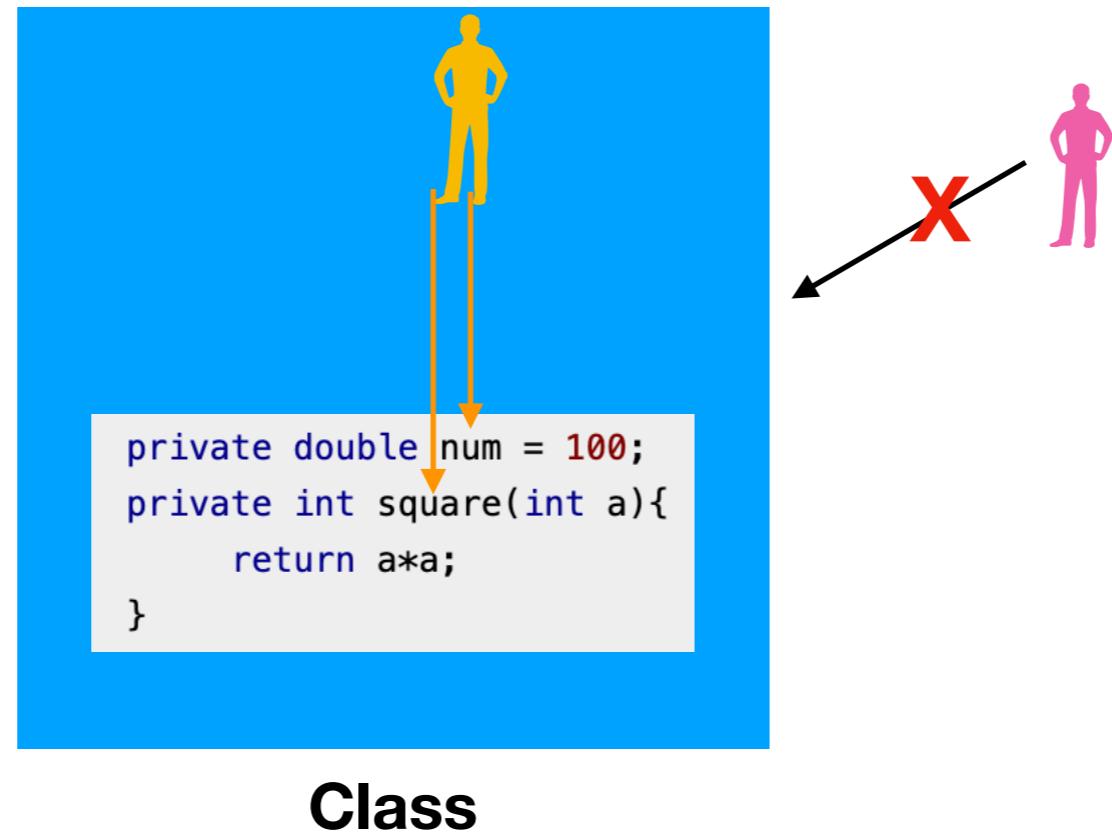
If you do not type anything for access modifier,
Java will accept it as default modifier

3) protected

4) public



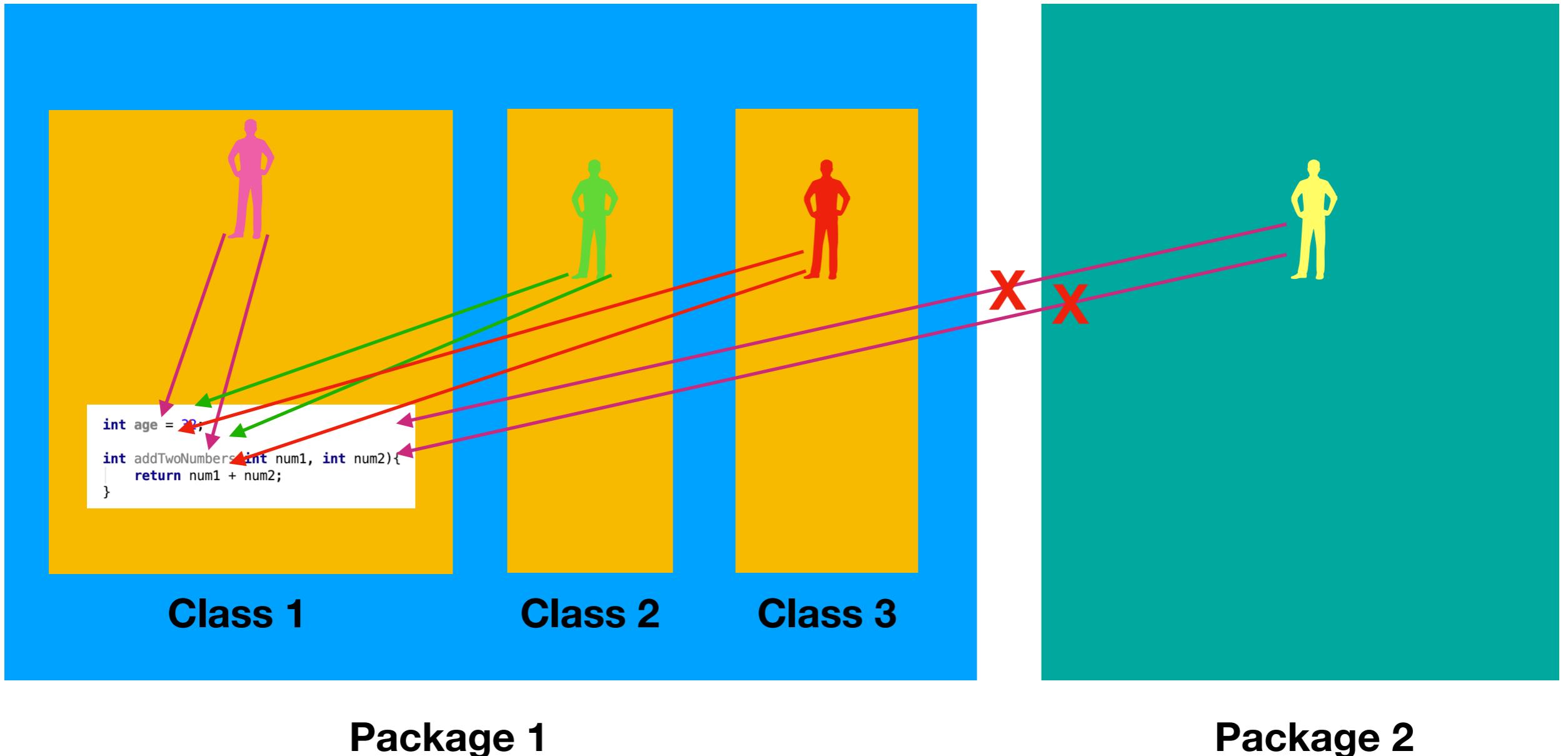
“private” Access Modifier



A class cannot be private



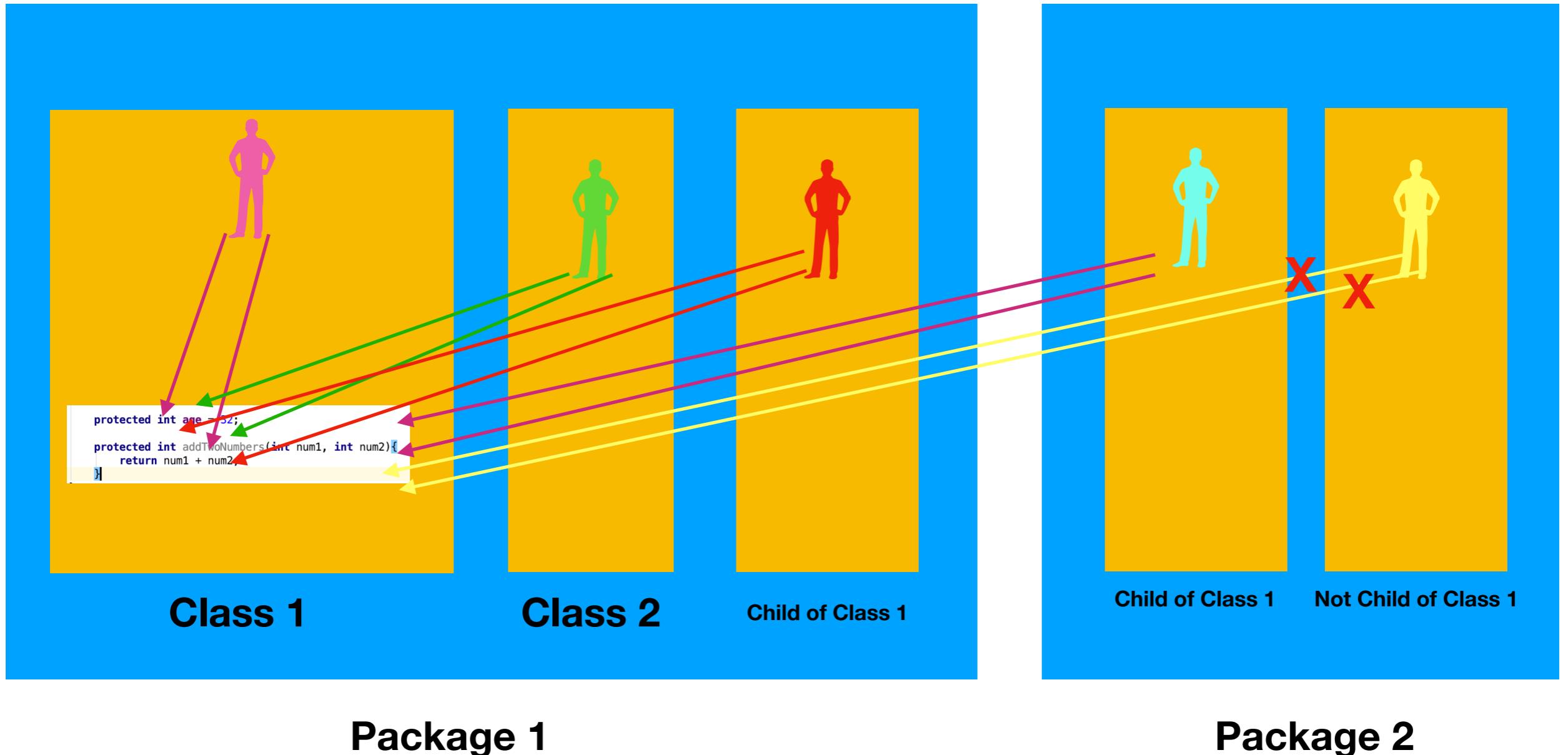
“default”(package private) Access Modifier



When we do not mention any access modifier, it is called default access modifier.



“protected” Access Modifier



Package 1

Package 2

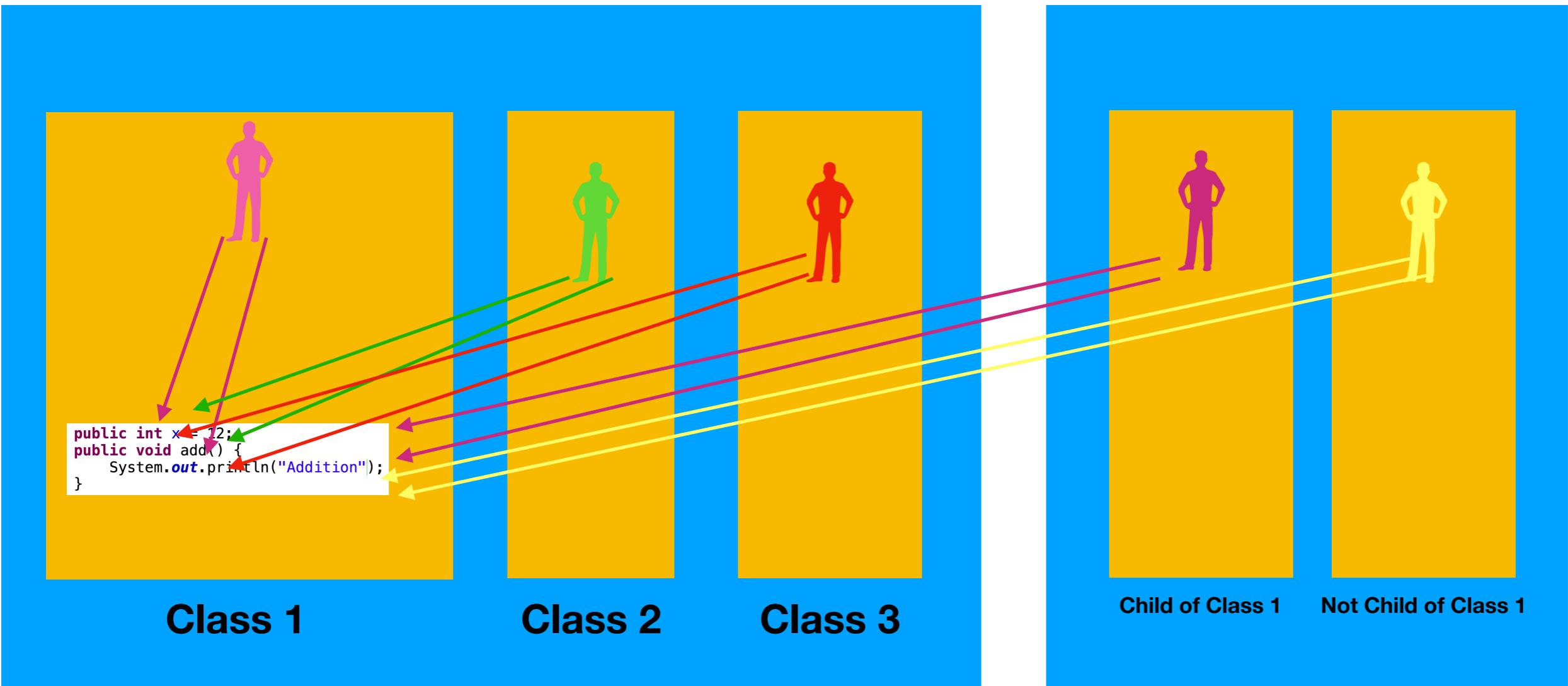
Classes cannot be declared as protected.



“public” Access Modifier

The data members, methods and classes that are declared public **can be accessed from anywhere**.

“public” access modifier **doesn’t put any restriction** on the access.



Package 1

Package 2



String Builder

- 1) `StringBuilder sb1 = new StringBuilder();` ==> To create a **StringBuilder** containing an empty sequence of characters
- 2) `StringBuilder sb2 = new StringBuilder("animal");` ==> To create a **StringBuilder** containing a specific value
- 3) `StringBuilder sb3 = new StringBuilder(5);` ==> Tells Java that we have some idea of how big the eventual value will be and would like the **StringBuilder** to reserve a certain number of slots for characters

Note: **StringBuffer** is very similar to **StringBuilder**. **StringBuilder** is faster than **StringBuffer**. If you will use multi-thread use **StringBuffer**

```
StringBuilder sb = new StringBuilder(5);
```

0	1	2	3	4

```
sb.append("anim");
```

a	n	i	m	
0	1	2	3	4

```
sb.append("als");
```

a	n	i	m	a	l	s		
0	1	2	3	4	5	6	7	...



StringBuilder Methods

```
StringBuilder sb = new StringBuilder("animals");
```

- 1) String sub1 = sb.substring(3);
System.out.println(sub1);

- 2) String sub2 = sb.substring(2, 5);
System.out.println(sub2);

- 3) sb.indexOf("n")

- 4) int lng = sb.length();
System.out.println(lng);

- 5) char ch = sb.charAt(6);
System.out.println(ch);



6) StringBuilder sb1 = new StringBuilder();

sb1.append("A").append("b");

System.out.println(sb1); // Ab

StringBuilder sb2 = new StringBuilder().append("A");

sb2.append("b").append("c");

System.out.println(sb2); // Abc



7) **StringBuilder sb1 = new StringBuilder("animal");**

sb1.insert(0,"X");

System.out.println(sb1); // Xanimal

sb1.insert(7,"X");

System.out.println(sb1); // XanimalX

sb1.insert(4,"X");

System.out.println(sb1); // XaniXmalX



8) StringBuilder sb1 = new StringBuilder("abcdef");

```
sb1.delete(1, 3);
System.out.println(sb1); // adef
```

9) StringBuilder sb2 = new StringBuilder("abcdef");

```
sb2.deleteCharAt(2);
System.out.println(sb2); // abed

sb2.deleteCharAt(4); // throws an exception
```



10) StringBuilder sb1 = new StringBuilder("abc");

```
sb1.reverse();
System.out.println(sb1); // cba
```

11) StringBuilder sb2 = new StringBuilder("abc");

```
sb2.toString();
System.out.println(sb2); // abc
```

Note: `StringBuilder` was added to Java in Java 5.

If you come across older code, you will see `StringBuffer` used for this purpose.
`StringBuffer` does the same thing but more slowly; therefore, use `StringBuilder`



Questions:

1)

What is the result of the following code?

```
7: StringBuilder sb = new StringBuilder();  
8: sb.append("aaa").insert(1, "bb").insert(4, "ccc");  
9: System.out.println(sb);
```

- A. abbaaccc
- B. abbaccca
- C. bbaaaccc
- D. bbaaccca
- E. An exception is thrown.
- F. The code does not compile.



2)

What is the result of the following code?

```
2: String s1 = "java";
3: StringBuilder s2 = new StringBuilder("java");
4: if (s1 == s2)
5:     System.out.print("1");
6: if (s1.equals(s2))
7:     System.out.print("2");
```

- A.** 1
- B.** 2
- C.** 12
- D.** No output is printed.
- E.** An exception is thrown.
- F.** The code does not compile.



3)

Which are the results of the following code? (Choose all that apply)

```
String numbers = "012345678";  
System.out.println(numbers.substring(1, 3));  
System.out.println(numbers.substring(7, 7));  
System.out.println(numbers.substring(7));
```

- A.** 12
- B.** 123
- C.** 7
- D.** 78
- E.** A blank line.
- F.** An exception is thrown.
- G.** The code does not compile.



4) What is the result of the following code?

```
4: int total = 0;  
5: StringBuilder letters = new StringBuilder("abcdefg");  
6: total += letters.substring(1, 2).length();  
7: total += letters.substring(6, 6).length();  
8: total += letters.substring(6, 5).length();  
9: System.out.println(total);
```

- A.** 1
- B.** 2
- C.** 3
- D.** 7
- E.** An exception is thrown.
- F.** The code does not compile.



- 5**) Which of the following compile? (Choose all that apply)
- A. `public void moreA(int... nums) {}`
 - B. `public void moreB(String values, int... nums) {}`
 - C. `public void moreC(int... nums, String values) {}`
 - D. `public void moreD(String... values, int... nums) {}`
 - E. `public void moreE(String[] values, ...int nums) {}`
 - F. `public void moreF(String... values, int[] nums) {}`
 - G. `public void moreG(String[] values, int[] nums) {}`



6) . Which are true of the following code? (Choose all that apply)

```
1: public class Rope {  
2:     public static void swing() {  
3:         System.out.print("swing ");  
4:     }  
5:     public void climb() {  
6:         System.out.println("climb ");  
7:     }  
8:     public static void play() {  
9:         swing();  
10:    climb();  
11: }  
12:    public static void main(String[] args) {  
13:        Rope rope = new Rope();  
14:        rope.play();  
15:        Rope rope2 = null;  
16:        rope2.play();  
17:    }  
18: }
```

- A. The code compiles as is.
- B. There is exactly one compiler error in the code.
- C. There are exactly two compiler errors in the code.
- D. If the lines with compiler errors are removed, the output is climb climb.
- E. If the lines with compiler errors are removed, the output is swing swing.
- F. If the lines with compile errors are removed, the code throws a NullPointerException.



7) What is the result of the following program?

```
1: public class Squares {  
2:     public static long square(int x) {  
3:         long y = x * (long) x;  
4:         x = -1;  
5:         return y;  
6:     }  
7:     public static void main(String[] args) {  
8:         int value = 9;  
9:         long result = square(value);  
10:        System.out.println(value);  
11:    } }
```

- A.** -1
- B.** 9
- C.** 81
- D.** Compiler error on line 9.
- E.** Compiler error on a different line.



Answers

- 5)** A, B, G. Options A and B are correct because the single vararg parameter is the last parameter declared. Option G is correct because it doesn't use any vararg parameters at all. Options C and F are incorrect because the vararg parameter is not last. Option D is incorrect because two vararg parameters are not allowed in the same method. Option E is incorrect because the ... for a vararg must be after the type, not before it.
- 6)** B, E. Line 10 does not compile because static methods are not allowed to call instance methods. Even though we are calling play() as if it were an instance method and an instance exists, Java knows play() is really a static method and treats it as such. If line 10 is removed, the code works. It does not throw a NullPointerException on line 16 because play() is a static method. Java looks at the type of the reference for rope2 and translates the call to Rope.play().
- 7)** B. Since Java is pass-by-value and the variable on line 8 never gets reassigned, it stays as 9. In the method square, x starts as 9. y becomes 81 and then x gets set to -1. Line 9 does set result to 81. However, we are printing out value and that is still 9.



MULTITHREADING is a process of executing **two or more threads simultaneously** to maximum utilization of CPU.



Thread 1: Image transfer

Thread 2: Voice transfer

Thread 3: Message transfer

Thread 4: Accepting call

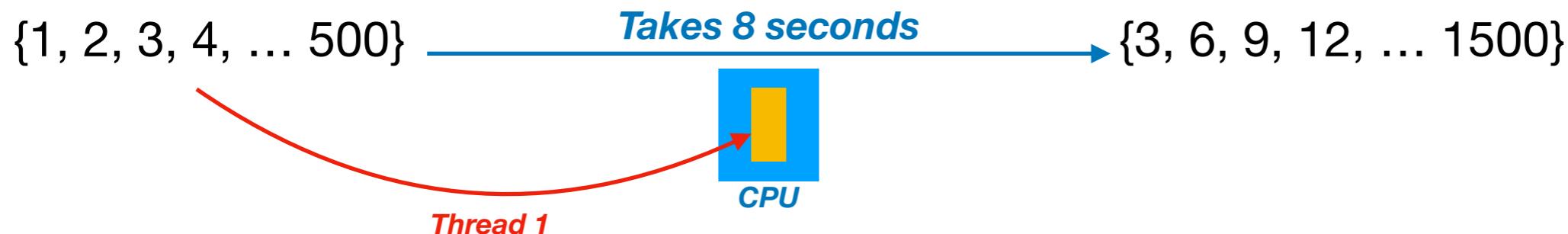
Thread 5: Sharing location

-
-
-
-

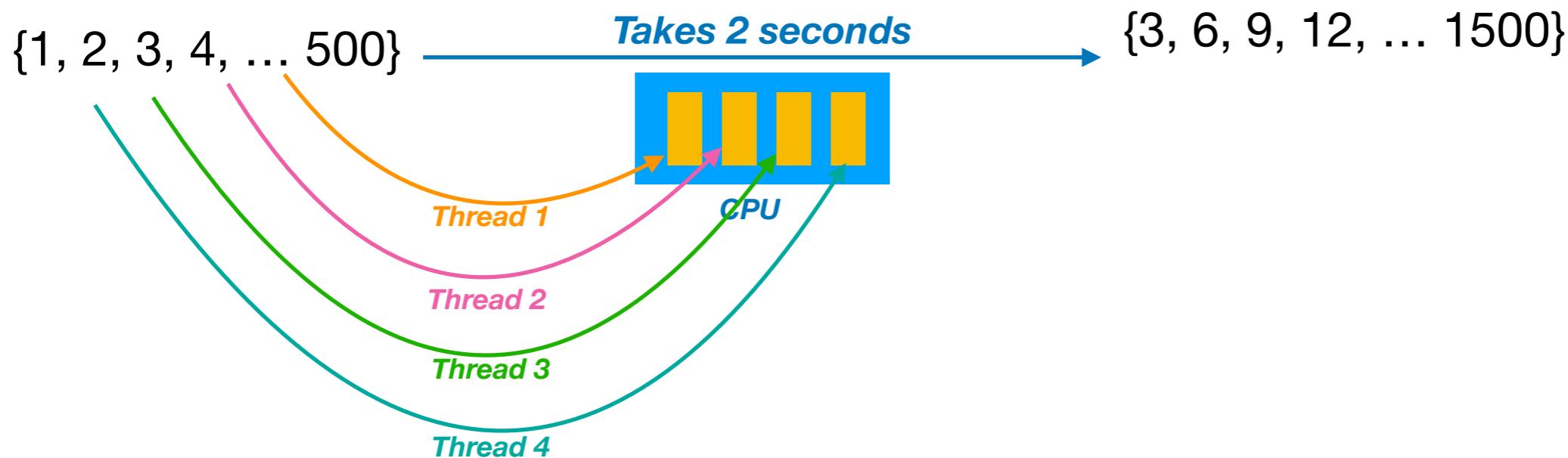


1. It saves time

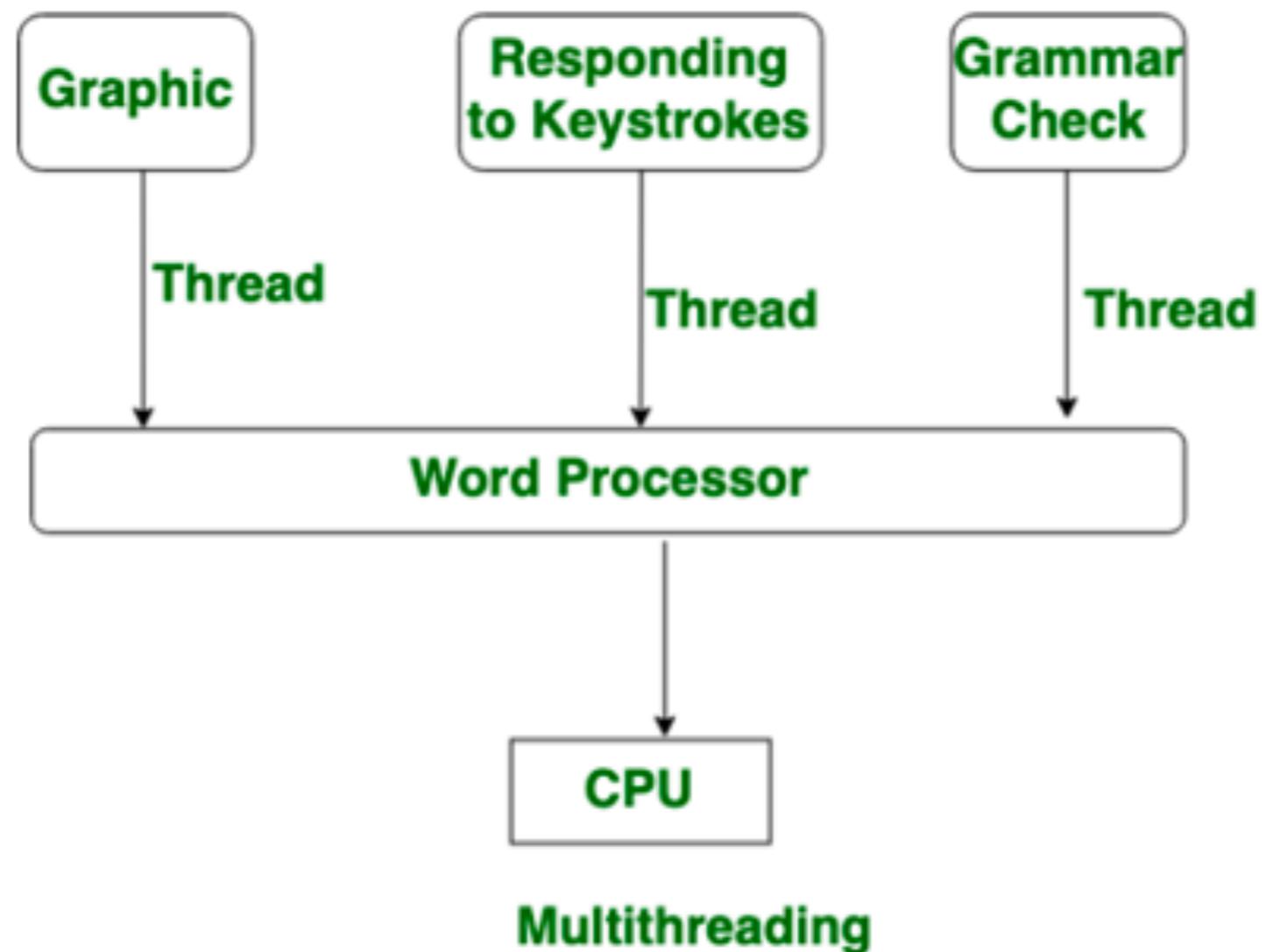
Suppose that multiplying all array elements by 3 will take 8 seconds



If you use multithreading it takes just 2 second

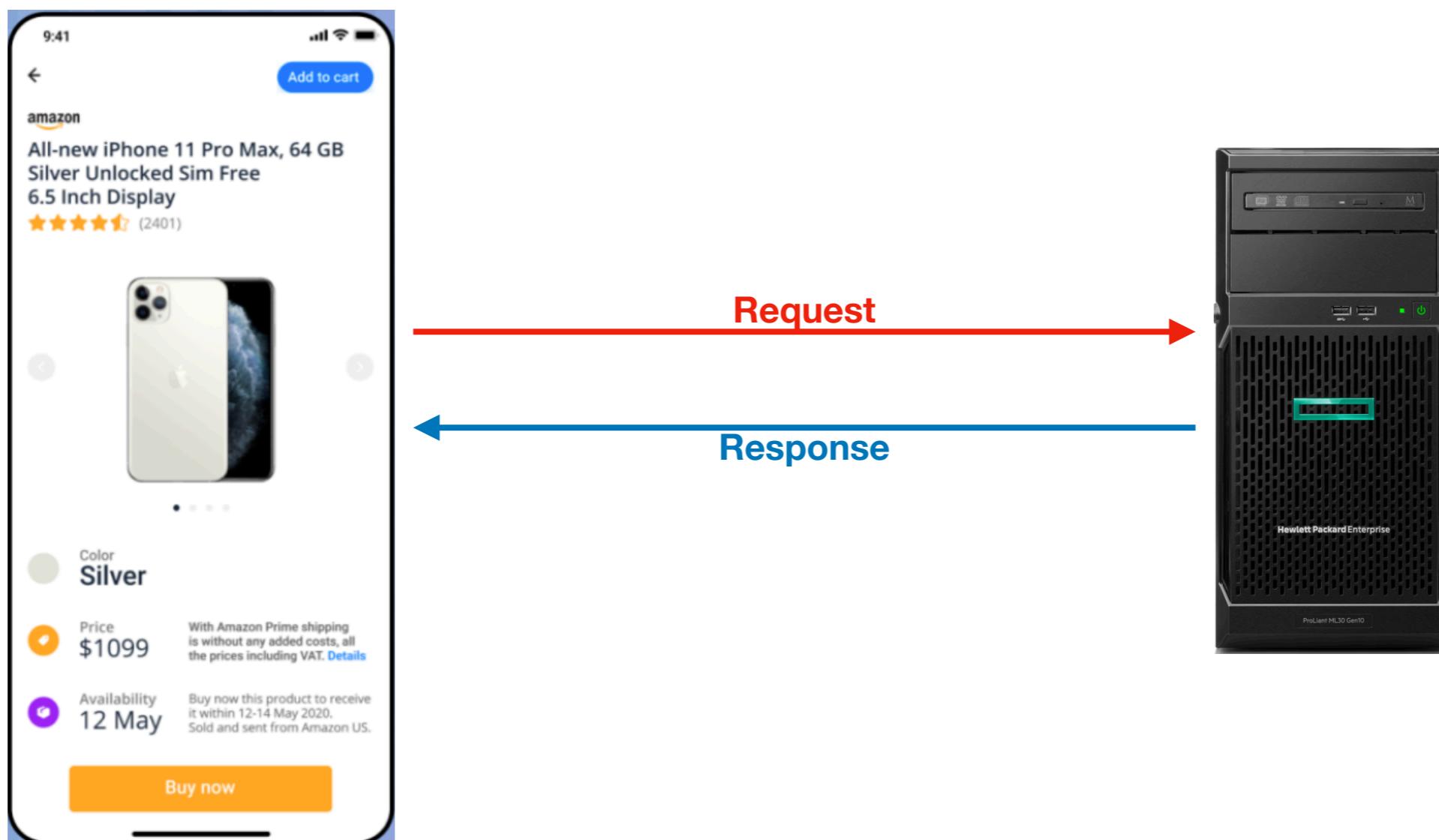


2. It is able to do multiple tasks at the same time



3. It prevents freezing of the application

Suppose that sending request and getting response will take 2 seconds. If you use single thread, your app freezes until the response come from the server, but if you use multithreading you can do another things on the app



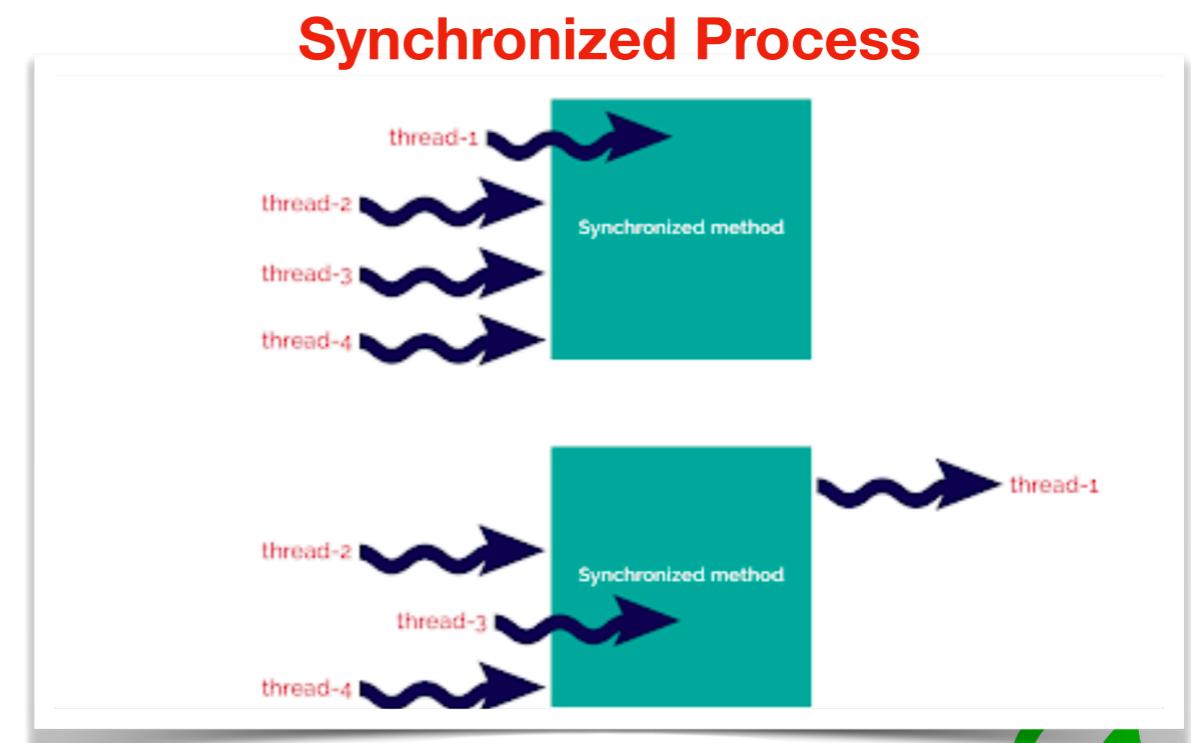
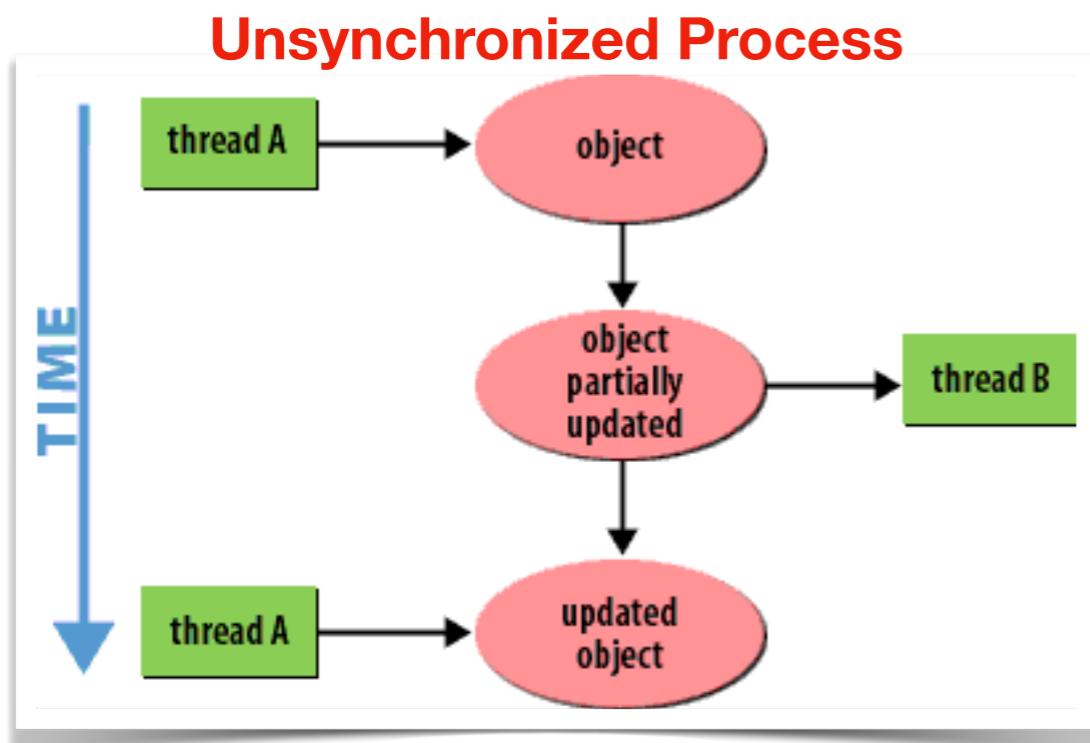
What is Synchronization?

Synchronization in java is the capability **to control the access** of multiple threads **to any shared resource**.

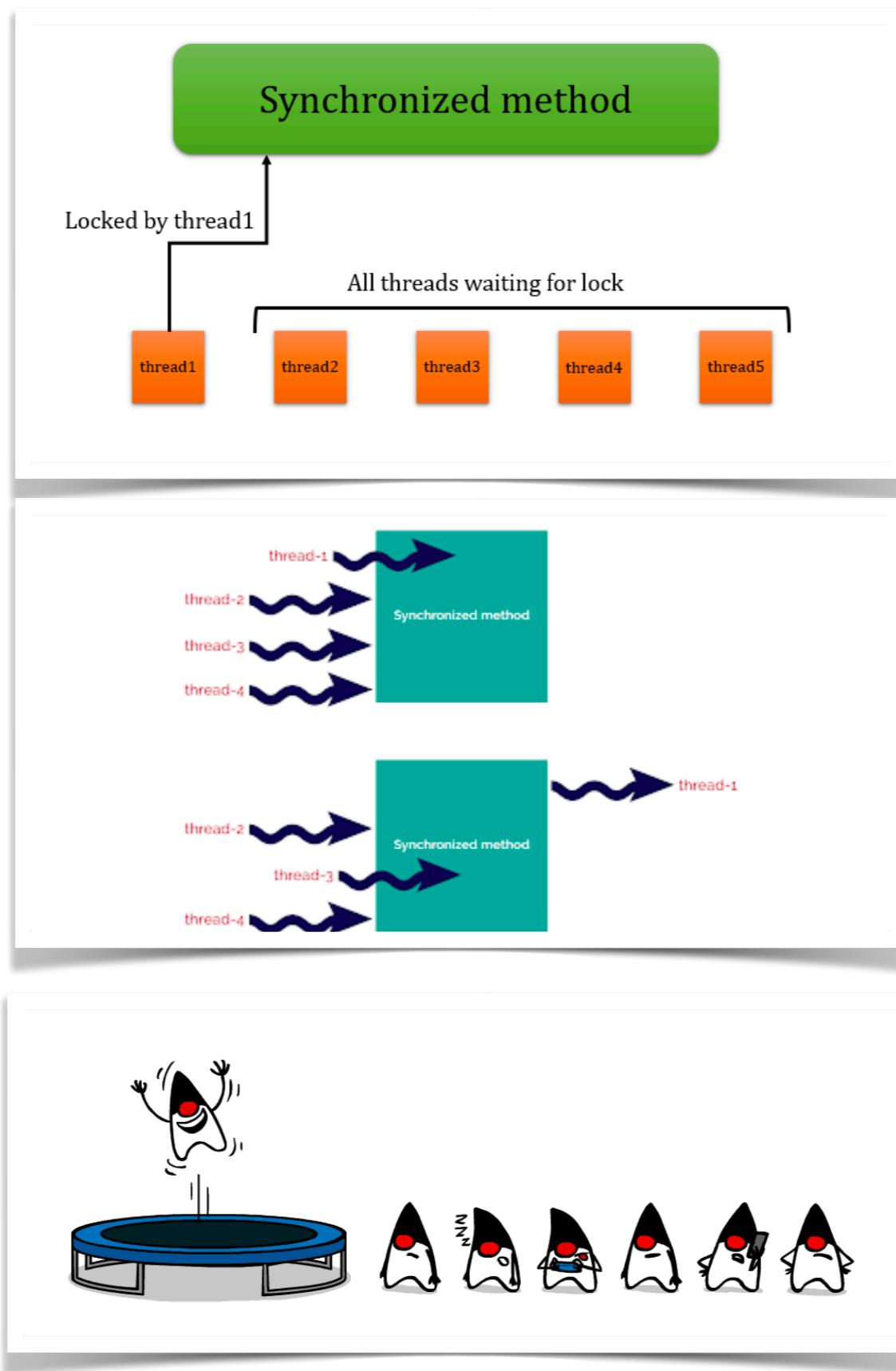
If you declare any method as synchronized, it is known as **synchronized method**.

Synchronized method is used **to lock an object for any shared resource**.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.



Java Thread Execution with Synchronized Method



Java Thread Execution with Unynchronized Method



String Class is not good to use in manipulation operations

Since **String** is *immutable* in Java, whenever we do String manipulation like concatenation, substring, etc. it generates a new String and discards the older String for garbage collection.

These are heavy operations and generate a lot of garbage in heap.

So Java has provided **StringBuffer** and **StringBuilder** classes that should be used **for string manipulation**.

StringBuffer and **StringBuilder** are *mutable* objects in Java. They provide append(), insert(), delete(), and substring() methods for String manipulation.



StringBuffer vs StringBuilder

StringBuffer was the only choice for String manipulation until Java 1.4. But, it has one **disadvantage** that all of its public methods are synchronized.

StringBuffer provides **Thread-Safety** but at a **performance cost**.

So Java 1.5 introduced a new class **StringBuilder**, which is similar to StringBuffer **except for Thread-Safety and Synchronization**.

If you are in a **single-threaded environment** or don't care about thread safety, you should use **StringBuilder**.

Otherwise, use **StringBuffer** for **Thread-Safe operations**.

Summary

If you want to operate a small amount of data with use “**String Class**”

Single-thread operation with a large amount of data under the string buffer use “**StringBuilder**”

Multithreaded operation of a large amount of data under the string buffer use “**StringBuffer**”



Encapsulation (Data Hiding)

The whole idea behind encapsulation is **to hide the implementation details** from users. If a data member is **private** it means it **can only be accessed within the same class**. No outside class can access private variable of the other classes.

However, if we setup **public getter** and **setter** methods to **read** and **update** the private data fields then the outside class can access those private data fields via public methods.

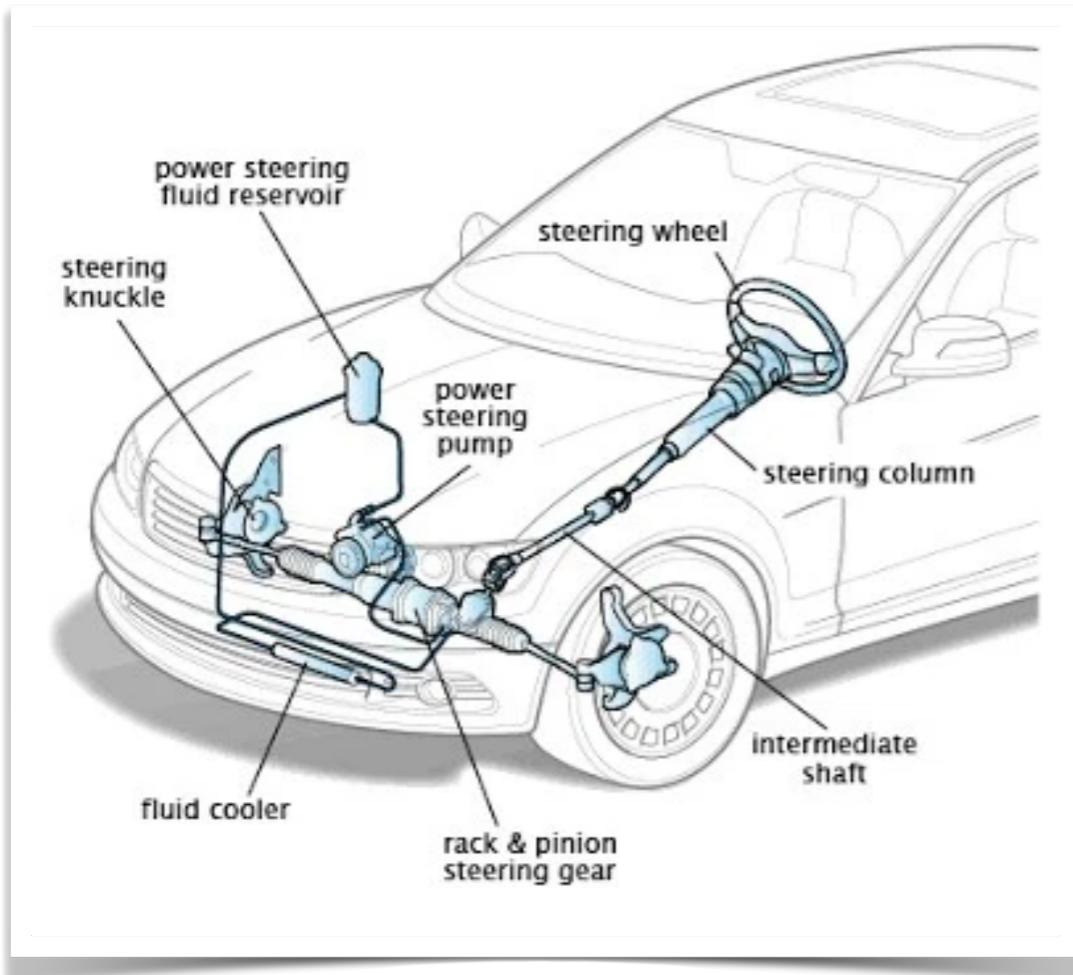
This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes.

That's why **encapsulation** is known as **data hiding**.

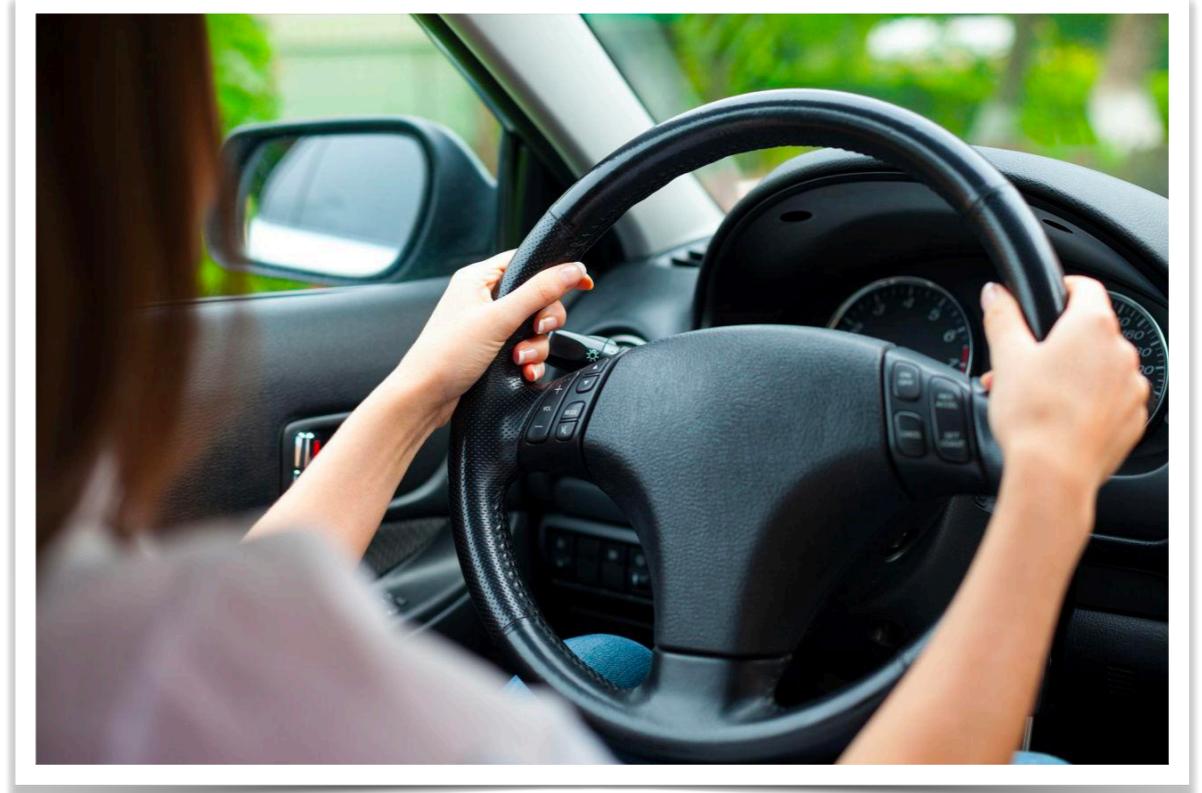
How to make encapsulation ?

- 1) Make the variables private
- 2) Create **public getter methods** to get the data value and/or create **public setter methods** to update the data value

Before Encapsulation (Data Hiding)



After Encapsulation (Data Hiding)



How to create getter method ?

```
public class CreditCards {  
  
    private String creditCardNums = "1234567891011213"; // Make the variable private  
  
    public String getCreditCardNums() { // Create getter method to get data value  
        return creditCardNums;  
    }  
  
}
```

Note 1: If you create just getter method, it means you created an immutable class. Because, you can just read the data values, you cannot update them.



How to create setter method ?

```
public class Account {  
  
    private int accountBalance$ = 12345;      // Make the variable private  
  
    public void setAccountBalance$ (int accountBalance$) { // Create setter method to update data value  
        this.accountBalance$ = accountBalance$;  
    }  
  
}
```

Note 2: If you use just setter method, it means you just can update the data value.
You cannot get the data value.



We can use getter and setter methods together

```
public class Account {  
  
    private int accountBalance$ = 12345;      // Make the variable private  
  
    public int getAccountBalance$ () {        // Create getter method to read data value  
        return accountBalance$;  
    }  
  
    public void setAccountBalance$ (int accountBalance$) {    // Create setter method to update data value  
        this.accountBalance$ = accountBalance$;  
    }  
  
}
```

Note 3: If you use getter and setter methods together, it means you can read and update the data values.



Encapsulation Task

- 1) Create class A**
- 2) Create class B**
- 3) Create a String, an int, and a boolean variable inside the class A and encapsulate them all**
- 4) Make String variable just readable**
- 5) Make int variable just updatable**
- 6) Make Boolean variable both readable and updatable**



Getter and setter methods are called as “Java Beans”.
We have some rules to name “Java Beans.”

1) Getter methods begin with “is” if the data type is boolean.

```
private boolean happy = true;  
  
public boolean isHappy() {  
    return happy;  
}
```

2) Getter methods begin with “get” if the data type is not boolean.

```
private int num = 123;  
  
public int getNum() {  
    return num;  
}
```

3) Setter methods begin with “set” for all data types

```
private String str = “Ali”;  
private boolean happy = true;  
  
public void setStr(String str) {  
    this.str = str;  
}  
  
public void setHappy(boolean happy) {  
    this.happy = happy;  
}
```



Questions

1)

Which are methods using JavaBeans naming conventions for accessors and mutators?
(Choose all that apply)

- A. public boolean getCanSwim() { return canSwim;}
- B. public boolean canSwim() { return numberWings;}
- C. public int getNumWings() { return numberWings;}
- D. public int numWings() { return numberWings;}
- E. public void setCanSwim(boolean b) { canSwim = b;}



2) Which of the following are true? (Choose all that apply)

- A.** Encapsulation uses package private instance variables.
- B.** Encapsulation uses private instance variables.
- C.** Encapsulation allows setters.
- D.** Immutability uses package private instance variables.
- E.** Immutability uses private instance variables.
- F.** Immutability allows setters.



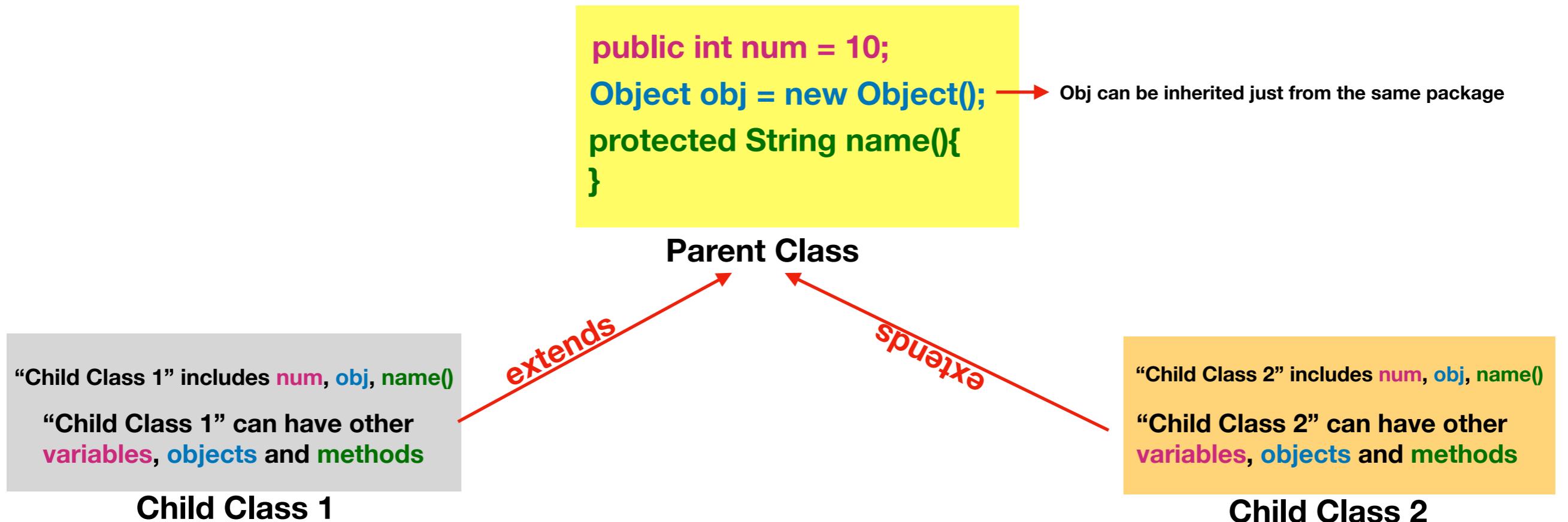
Answers

- 1) C, E. Option A is incorrect because the property is of type boolean and getters must begin with `is` for booleans. Options B and D are incorrect because they don't follow the naming convention of beginning with `get/is/set`. Options C and E follow normal getter and setter conventions.
- 2) B, C, E. Encapsulation requires using methods to get and set instance variables so other classes are not directly using them. Instance variables must be private for this to work. Immutability takes this a step further, allowing only getters, so the instance variables do not change state.



Inheritance

When creating a new class in Java, you can **define the class to inherit from an existing class**. Inheritance is the process by which the new child subclass automatically includes any **public or protected primitives, objects, or methods defined in the parent class**.



Note 1: Child classes can inherit just public and protected ones.

Note 2: Private ones cannot be inherited.

Note 3: All subclasses will inherit all default variables in the *same package only*.

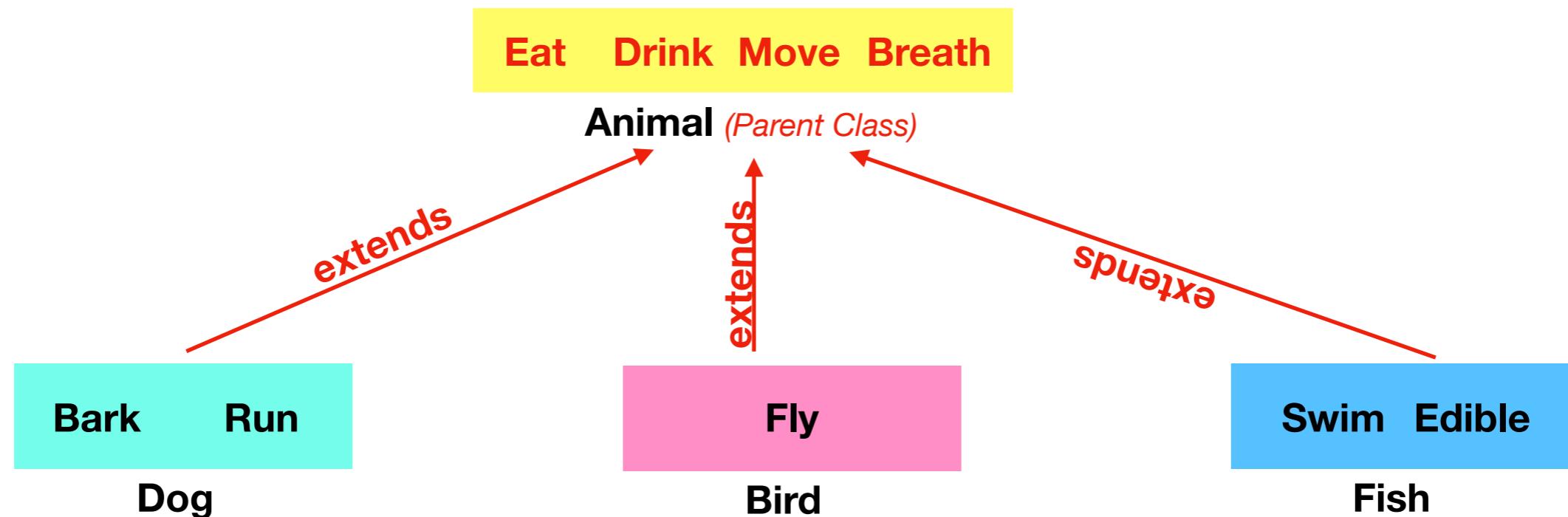
Subclasses outside the package will not inherit any default members.

Note 4: Static Methods or variables do not take part in inheritance.



Advantages of Inheritance

The aim of inheritance is to provide the **reusability** of code so that a class has to **write only the unique features** and rest of the common properties and functionalities can be extended from the another class.

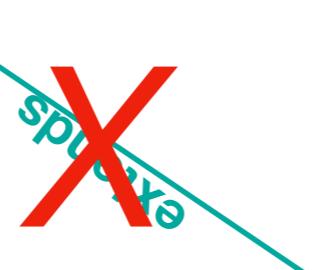


Single Inheritance: Java supports **single inheritance**, by which a class may **inherit** from **only one direct parent class**.

A CHILD CANNOT HAVE MORE THAN ONE PARENT

```
public int num = 10;  
Object obj = new Object();  
protected String  
name(){  
}
```

Parent Class 1



```
public int num = 10;  
Object obj = new Object();  
protected String  
name(){  
}
```

Parent Class 2



“Child Class 1” can use **num**, **obj**, **name()**

“Child Class 1” can have other
variables, **objects** and **methods**

Child Class 1

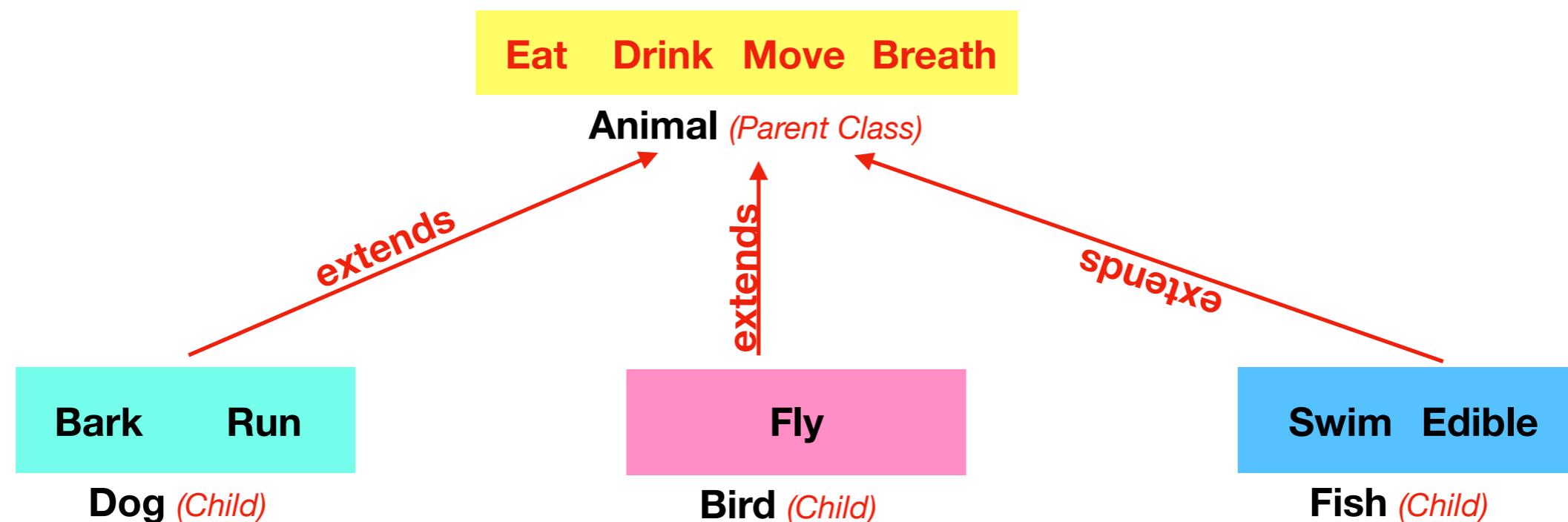


Hierarchical Inheritance: Refers to a child and parent class relationship where more than one classes extends the same class. For example, classes Dog, Bird, and Fish extends the same class Animal.

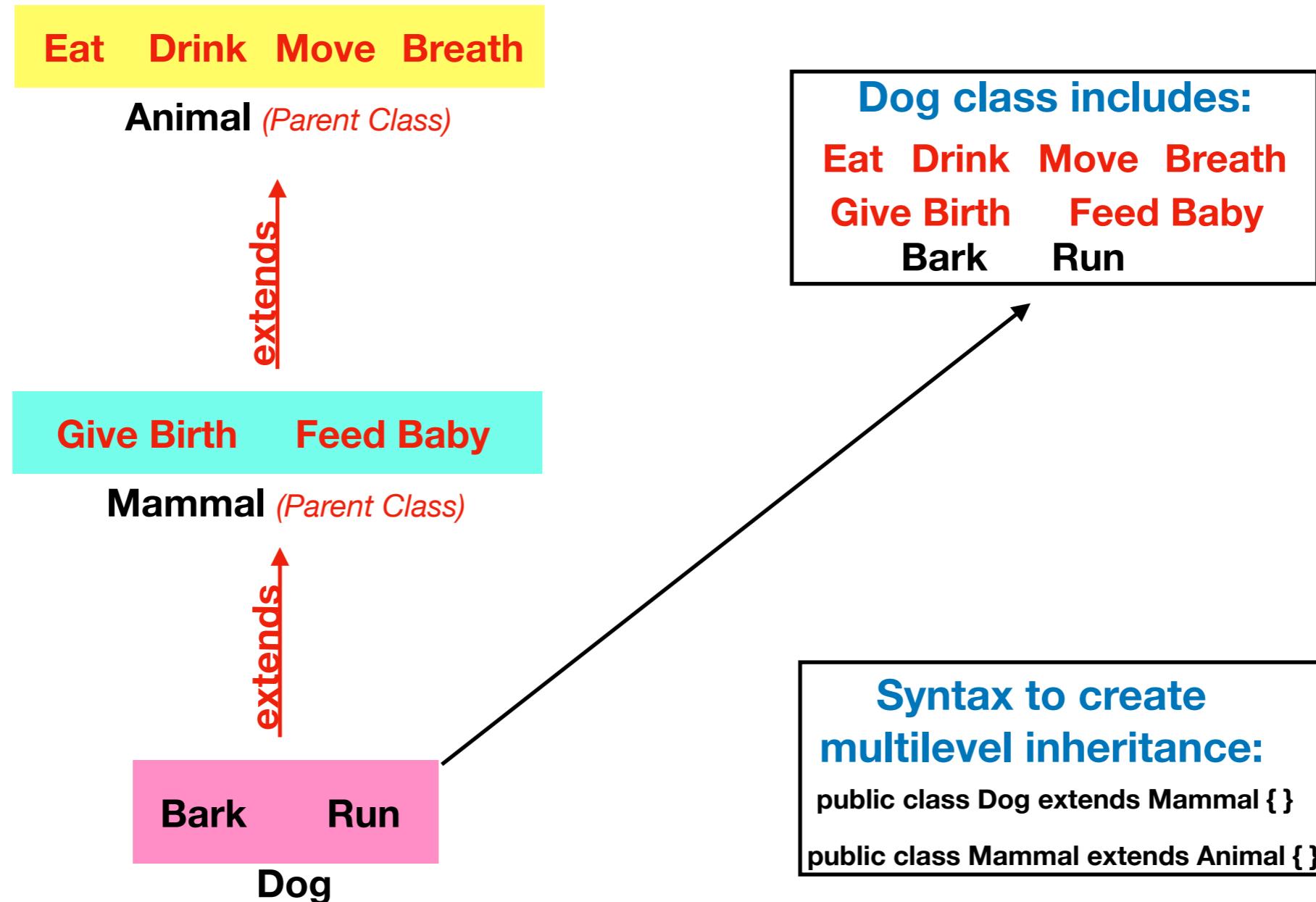
```
public class Dog extends Animal{}
```

```
public class Bird extends Animal{}
```

```
public class Fish extends Animal{}
```



Multilevel Inheritance: Refers to a child and parent class relationship where a class extends the child class. For example; class Dog extends class Animal, and class Animal extends class Mammal.



Multiple Inheritance: Refers to the concept of one class extending more than one classes, which means a child class has two parent classes.

For example; “Child Class” extends both classes “Parent Class 1” and “Parent Class 2”.

Java doesn't support multiple inheritance.

```
public int num1 = 10;  
Object obj1 = new Object();  
protected String name(){  
    System.out.println("Veli");  
}
```

Parent Class 1

```
public int num2 = 10;  
Object obj2 = new Object();  
protected String name(){  
    System.out.println("Ali");  
}
```

Parent Class 2

```
protected String name(){  
}
```

Child Class



From which parent ?



Review Questions

- 1) What is Encapsulation ?**
- 2) How do you hide data ?**
- 3) How do you access hidden data from other classes ?**
- 4) What does getter() method do ?**
- 5) What does setter() method do ?**
- 6) What is immutable class ?**
- 7) What is the naming convention for setter() method ?**
- 8) What is the naming convention for getter() method ?**



Review Questions with Answers

1) What is Encapsulation ?

Encapsulation is data hiding

2) How do you hide data ?

By using private access modifier

3) How do you access hidden data from other classes ?

By creating getter and setter methods

4) What does getter() method do ?

Getter method reads the data value

5) What does setter() method do ?

Setter method updates the data value

6) What is immutable class ?

If you use just getter method (no setter methods) in a class, it means you just read the data, you cannot update the data. That kind of classes are called immutable class

7) What is the naming convention for setter() method ?

For all data types we should start with set...

8) What is the naming convention for getter() method ?

If data type is boolean then getter methods name should start with is.



How to use inheritance ?

```
public class Animal{  
    boolean isBirth = true;  
  
    public void feed(){  
        System.out.println("I feed my baby");  
    }  
}
```

```
public class Mammal extends Animal{  
    boolean isFly = true;  
    String name = "Bat";  
  
    public void see(){  
        System.out.println("I cannot see");  
    }  
  
    public static void main(String args[]){  
        Mammal mammal = new Mammal();  
        mammal.see();  
        mammal.feed();  
        System.out.println(mammal.isBirth);  
        System.out.println(mammal.isFly);  
        System.out.println(mammal.name);  
    }  
}
```

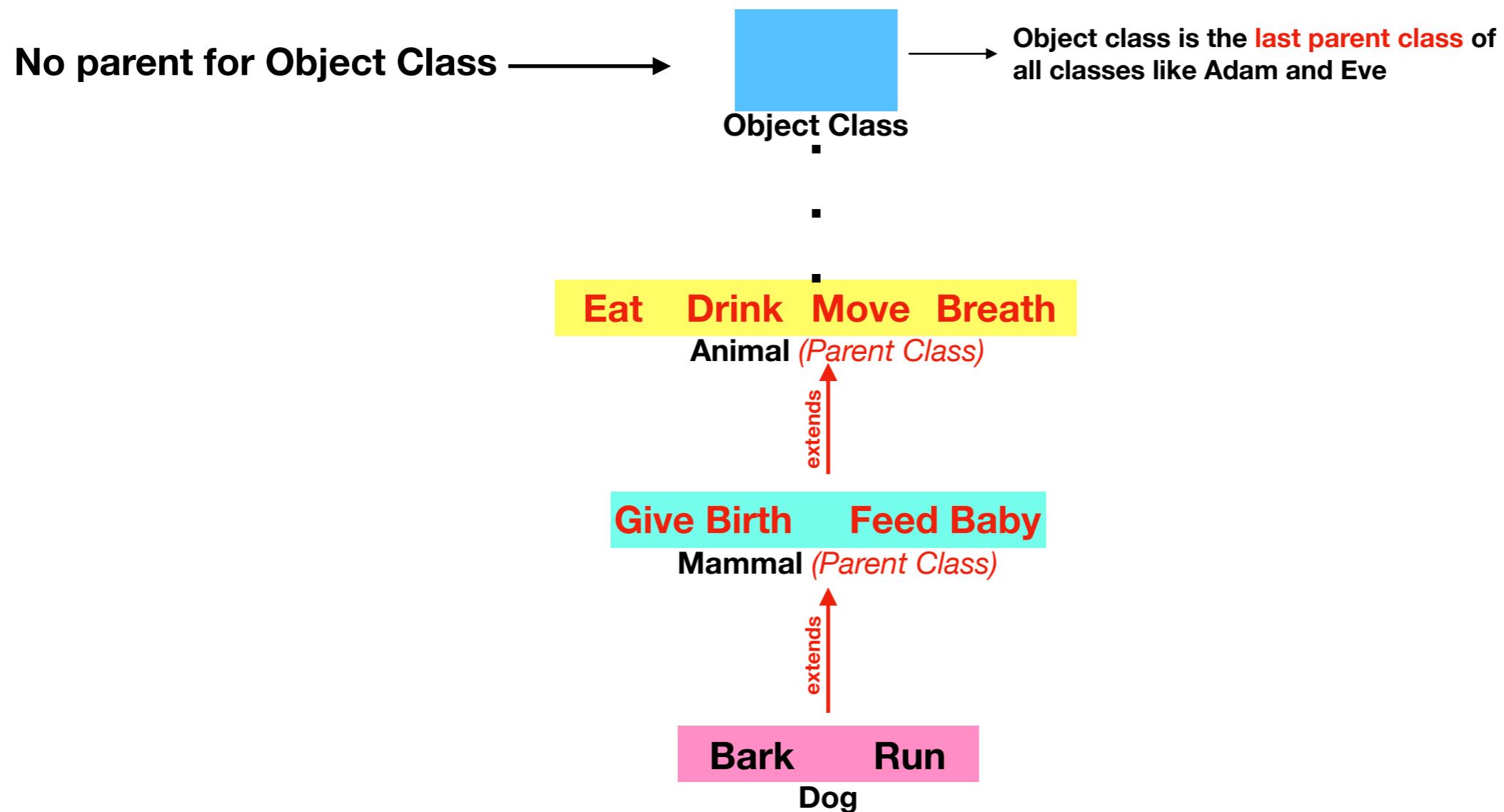
→ **extends keyword**

Mammal class has; **isFly, name, see(), isBirth, feed()**

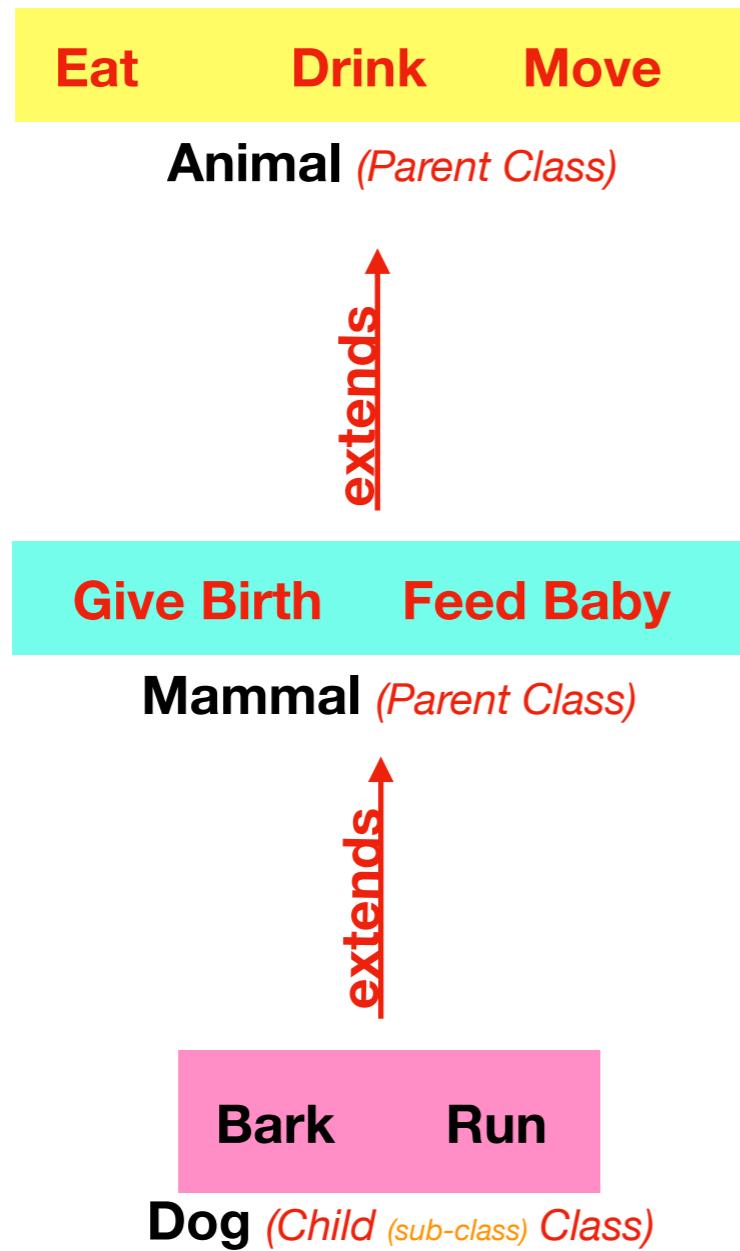


java.lang.Object Class

In Java, all classes inherit from a single class which is Object Class. Furthermore, Object is the only class that doesn't have any parent classes in Java.



IS-A (\uparrow) and HAS-A (\downarrow) Relationship



- Dog IS-A Mammal ==> True**
- Dog IS-A Animal ==> True**
- Mammal IS-A Animal ==> True**
- Animal IS-A Dog ==> False**
- Animal IS-A Mammal ==> False**
- Mammal IS-A Dog ==> False**
- Mammal HAS-A Dog ==> True**
- Animal HAS-A Mammal ==> True**
- Animal HAS-A Dog ==> True**
- Dog HAS-A Animal ==> False**
- Mammal HAS-A Animal ==> False**



Calling Constructors Rules in Inheritance

1) Every time parent constructor runs first

```
public class Animal {  
    public Animal() {  
        System.out.println("Parent constructor runs first");  
    }  
}
```

```
public class Zebra extends Animal {  
    public Zebra() {  
        System.out.println("Child constructor runs at the end");  
    }  
}
```

```
public class Animal {  
    public Animal() {  
        System.out.println("Parent constructor runs first");  
    }  
}
```

```
public class Zebra extends Animal {  
    public Zebra() {  
        super();  
        System.out.println("Child constructor runs at the end");  
    }  
}
```

Note 1: To call default constructor use **super()** keyword or not. Both work.



2) If there is no appropriate parent constructor, then the code does not compile even child class has constructor.

```
public class Animal {  
    public Animal(int age) {  
        System.out.println("Parent constructor with parameter");  
    }  
}
```

```
public class Zebra extends Animal {  
    public Zebra() {  
        super();  
        System.out.println("Child constructor");  
    }  
}
```

Does not compile

```
public class Animal {  
    public Animal(int age) {  
        System.out.println("Parent constructor with parameter");  
    }  
}
```

```
public class Zebra extends Animal {  
    public Zebra() {  
        super(5);  
        System.out.println("Child constructor");  
    }  
}
```

Compiles



3) **super()** keyword is used to call parent constructor, **this()** keyword is used to call in class constructor.

What is the output ?

```
public class Animal{  
    public Animal(){  
        System.out.println("Parent Constructor");  
    }  
}  
  
class Mammal extends Animal{  
    public Mammal(int age){  
        super();  
        System.out.println("Child Constructor called by this()");  
    }  
    public Mammal(){  
        this( age: 11);  
        System.out.println("Child Constructor");  
    }  
    public static void main(String args[]){  
        Mammal mammal = new Mammal();  
    }  
}
```



What is the output ?

```
public class Animal{  
    public Animal(){  
        System.out.println("Parent Constructor");  
    }  
}  
  
class Mammal extends Animal{  
    public Mammal(int age){  
        super();  
        System.out.println("Child Constructor called by this()");  
    }  
    public Mammal(){  
        this( age: 11);  
        System.out.println("Child Constructor");  
    }  
    public static void main(String args[]){  
        Mammal mammal = new Mammal( age: 15);  
    }  
}
```



3) **super()** and **this()** keywords must be the first statement of the constructor, not the second statement.

```
public class Animal {  
    public Animal(int age) {  
        System.out.println("Parent constructor runs first");  
    }  
}
```

```
public class Zebra extends Animal {  
    public Zebra() {  
        System.out.println("Child constructor runs at the end");  
        super();  
    }  
}
```

Does not compile

```
public class Animal {  
    public Animal() {  
        System.out.println("Parent constructor runs first");  
    }  
}
```

```
public class Zebra extends Animal {  
    public Zebra() {  
        super();  
        System.out.println("Child constructor runs at the end");  
    }  
}
```

Compiles

Note 2: On the left **super()** keyword is not the first statement inside the constructor; therefore, you will get compile time error.

Note 3: **super()** and **this()** keywords cannot be in a constructor at the same time.
super and **this** keywords can be in a constructor at the same time.

Note 4: **super()** and **this()** are used to call constructors.
super and **this** are used to call variables.



These three classes are completely same !!!

**1) public class Donkey {
}**

The class has no any constructor, but Java gives a default constructor
**public Donkey() {
}**

**2) public class Donkey {
 **public Donkey() {
 }
}****

**3) public class Donkey {
 **public Donkey() {
 super();
 }
}****

super(); calls the constructor from Object class. The constructor
inside object class is
**public Donkey() {
}**



Calling Inherited Class Members

Note: “super” keyword is used to call data from parent class. “this” keyword is used to call in class data

Note: Actually, you can use “this” to call parent data; however, it is not recommended. Because, if some data has same names in child and parent classes, “this” cannot call the parent data.

```
public class Animal {  
  
    protected int num1 = 10;  
    public String name = "Ali";  
    private int num2 = 11;  
  
    public int getNum2(){  
        return num2;  
    }  
  
    public void setNum2(int num2){  
        this.num2 = num2;  
    }  
}
```



```
public class Mammal extends Animal {  
  
    protected int num3 = 12;  
    public String name2 = "Veli";  
    private int num4 = 13;  
  
    public Mammal(){  
        System.out.println(this.num1);  
        System.out.println(super.num1);  
  
        System.out.println(this.getNum2());  
        System.out.println(super.getNum2());  
  
        this.setNum2(23);  
        System.out.println(this.getNum2());  
        System.out.println(super.getNum2());  
  
        super.setNum2(33);  
        System.out.println(this.getNum2());  
        System.out.println(super.getNum2());  
  
        System.out.println(this.num3);  
        System.out.println(this.num4);  
  
        System.out.println(this.name);  
        System.out.println(super.name);  
  
        System.out.println(this.name2);  
    }  
  
    public static void main(String args[]){  
        Mammal mammal = new Mammal();  
    }  
}
```



Certificate Preparation

1)

What is the output of the following application?

```
1: public class CompareValues {  
2:     public static void main(String[] args) {  
3:         int x = 0;  
4:         while(x++ < 10) {}  
5:         String message = x > 10 ? "Greater than" : false;  
6:         System.out.println(message+","+x);  
7:     }  
8: }
```

- A. Greater than,10
- B. false,10
- C. Greater than,11
- D. false,11
- E. The code will not compile because of line 4.
- F. The code will not compile because of line 5.



2)

What is the output of the following code snippet?

```
3: java.util.List<Integer> list = new java.util.ArrayList<Integer>();  
4: list.add(10);  
5: list.add(14);  
6: for(int x : list) {  
7:     System.out.print(x + ", ");  
8:     break;  
9: }
```

- A.** 10, 14,
- B.** 10, 14
- C.** 10,
- D.** The code will not compile because of line 7.
- E.** The code will not compile because of line 8.
- F.** The code contains an infinite loop and does not terminate.



3)

What is the output of the following code snippet?

```
3: int x = 4;  
4: long y = x * 4 - x++;  
5: if(y<10) System.out.println("Too Low");  
6: else System.out.println("Just right");  
7: else System.out.println("Too High");
```

- A. Too Low
- B. Just Right
- C. Too High
- D. Compiles but throws a NullPointerException.
- E. The code will not compile because of line 6.
- F. The code will not compile because of line 7.



4)

What is the output of the following code?

```
1: public class TernaryTester {  
2:     public static void main(String[] args) {  
3:         int x = 5;  
4:         System.out.println(x > 2 ? x < 4 ? 10 : 8 : 7);  
5:     } }
```

- A.** 5
- B.** 4
- C.** 10
- D.** 8
- E.** 7
- F.** The code will not compile because of line 4.



5)

How many times will the following code print "Hello World"?

```
3: for(int i=0; i<10 ; ) {  
4:     i = i++;  
5:     System.out.println("Hello World");  
6: }
```

- A. 9
- B. 10
- C. 11
- D. The code will not compile because of line 3.
- E. The code will not compile because of line 5.
- F. The code contains an infinite loop and does not terminate.



6)

What is the output of the following code?

```
1: public class ArithmeticSample {  
2:     public static void main(String[] args) {  
3:         int x = 5 * 4 % 3;  
4:         System.out.println(x);  
5:     } } 
```

- A. 2
- B. 3
- C. 5
- D. 6
- E. The code will not compile because of line 3.



7) What is the output of the following code snippet?

```
3: int x = 0;  
4: String s = null;  
5: if(x == s) System.out.println("Success");  
6: else System.out.println("Failure");
```

- A. Success
- B. Failure
- C. The code will not compile because of line 4.
- D. The code will not compile because of line 5.



8) What is the output of the following code snippet?

```
3: int x1 = 50, x2 = 75;  
4: boolean b = x1 >= x2;  
5: if(b = true) System.out.println("Success");  
6: else System.out.println("Failure");
```

- A.** Success
- B.** Failure
- C.** The code will not compile because of line 4.
- D.** The code will not compile because of line 5.



9) What is the output of the following code snippet?

```
3: do {  
4:     int y = 1;  
5:     System.out.print(y++ + " ");  
6: } while(y <= 10);
```

- A.** 1 2 3 4 5 6 7 8 9
- B.** 1 2 3 4 5 6 7 8 9 10
- C.** 1 2 3 4 5 6 7 8 9 10 11
- D.** The code will not compile because of line 6.
- E.** The code contains an infinite loop and does not terminate.



10)

What is the result of the following code snippet?

```
3: int m = 9, n = 1, x = 0;  
4: while(m > n) {  
5:     m--;  
6:     n += 2;  
7:     x += m + n;  
8: }  
9: System.out.println(x);
```

- A.** 11
- B.** 13
- C.** 23
- D.** 36
- E.** 50
- F.** The code will not compile because of line 7.



Answers of Certificate Preparation Questions

1) F. In this example, the ternary operator has two expressions, one of them a String and the other a boolean value. The ternary operator is permitted to have expressions that don't have matching types, but the key here is the assignment to the String reference. The compiler knows how to assign the first expression value as a String, but the second boolean expression cannot be set as a String; therefore, this line will not compile.

2) C. This code does not contain any compilation errors or an infinite loop, so options D, E, and F are incorrect. The break statement on line 8 causes the loop to execute once and finish, so option C is the correct answer.

3) F. The code does not compile because two else statements cannot be chained together without additional if-then statements, so the correct answer is option F. Option E is incorrect as Line 6 by itself does not cause a problem, only when it is paired with Line 7. One way to fix this code so it compiles would be to add an if-then statement on line 6. The other solution would be to remove line 7.

4) D. As you learned in the section “Ternary Operator,” although parentheses are not required, they do greatly increase code readability, such as the following equivalent statement:

```
System.out.println((x > 2) ? ((x < 4) ? 10 : 8) : 7)
```

We apply the outside ternary operator first, as it is possible the inner ternary expression may never be evaluated. Since $x > 2$ is true, this reduces the problem to:

```
System.out.println((x < 4) ? 10 : 8)
```

Since x is greater than 2, the answer is 8, or option D in this case.



- 5) F.** In this example, the update statement of the for loop is missing, which is fine as the statement is optional, so option D is incorrect. The expression inside the loop increments *i* but then assigns *i* the old value. Therefore, *i* ends the loop with the same value that it starts with: 0. The loop will repeat infinitely, outputting the same statement over and over again because *i* remains 0 after every iteration of the loop.
- 6) A.** The * and % have the same operator precedence, so the expression is evaluated from left-to-right. The result of $5 * 4$ is 20, and $20 \% 3$ is 2 (20 divided by 3 is 18, the remainder is 2). The output is 2 and option A is the correct answer.
- 7) D.** The variable *x* is an int and *s* is a reference to a String object. The two data types are incomparable because neither variable can be converted to the other variable's type. The compiler error occurs on line 5 when the comparison is attempted, so the answer is option D.
- 8) A.** The code compiles successfully, so options C and D are incorrect. The value of *b* after line 4 is false. However, the if-then statement on line 5 contains an assignment, not a comparison. The variable *b* is assigned true on line 3, and the assignment operator returns true, so line 5 executes and displays Success, so the answer is option A.



- 9)** D. The variable *y* is declared within the body of the do-while statement, so it is out of scope on line 6. Line 6 generates a compiler error, so option D is the correct answer.
- 10)** D. Prior to the first iteration, *m* = 9, *n* = 1, and *x* = 0. After the iteration of the first loop, *m* is updated to 8, *n* to 3, and *x* to the sum of the new values for *m* + *n*, $0 + 11 = 11$. After the iteration of the second loop, *m* is updated to 7, *n* to 5, and *x* to the sum of the new values for *m* + *n*, $11 + 12 = 23$. After the iteration of the third loop, *m* is updated to 6, *n* to 7, and *x* to the sum of the new values for *m* + *n*, $23 + 13 = 36$. On the fourth iteration of the loop, *m* > *n* evaluates to false, as $6 < 7$ is not true. The loop ends and the most recent value of *x*, 36, is output, so the correct answer is option D.



Review Questions

- 1) What are the advantages of “**inheritance**” ?**
- 2) What is the syntax to create a parent class for a class ?**
- 3) Which access modifiers can be inherited ?**
- 4) What is the difference between **super()** and **this()** ?**
- 5) What is the difference between **super()** and **super** ?**
- 6) What is the difference between **this()** and **this** ?**
- 7) What is the difference between **super** and **this** ?**
- 8) **super()** and **this()** must be the first statement in a constructor. **True or False****
- 9) **super()** and **this()** must be used just once in a constructor. **True or False****
- 10) **super()** and **this()** can be used together at the same time in a constructor.
True or False**



Review Questions with Answers

1) What are the advantages of “**inheritance**” ?

1) Reusability 2) Maintenance 3) Less code

2) What is the syntax to create a parent class for a class ?

public class Child extends Parent { }

3) Which access modifiers can be inherited ?

public and protected class members can be inherited, private ones cannot.
default ones can be if they are in the same package

4) What is the difference between **super()** and **this()** ?

super() is used to call parent constructor, this() is used to call in class constructor.

5) What is the difference between **super()** and **super** ?

super() is used to call parent constructor, super is used to call parent class variables and methods.

6) What is the difference between **this()** and **this** ?

this() is used to call in class constructor, this is used to call in class variables and methods.

7) What is the difference between **super** and **this** ?

super is used to call parent variables and methods, this is used to call in class variables and methods.
You can use this to call parent variables and methods as well but it is not recommended, because if the variable or method names are same in parent and child classes, it can cause problem.

8) **super()** and **this()** must be the first statement in a constructor. True or False

9) **super()** and **this()** must be used just once in a constructor. True or False

10) **super()** and **this()** can be used together at the same time in a constructor.

True or False



Method Signature

The method signature consists of the **method name** and the **parameter list**.

- 1) If you decrease or increase the number of the parameters, it means you changed the signature. You have two different methods.

```
public String add(String a, int b){  
    String c = a+b;  
    return c;  
}  
private int add(String y, int x){  
    int c = x;  
    return c;  
}  
public static void main(String args[]){  
}
```

```
class Example {  
  
    public String add(String a, int b){  
        String c = a+b;  
        return c;  
    }  
    private int add(int y){  
        int c = y;  
        return c;  
    }  
    public static void main(String args[]){  
    }  
}
```



**2) If you change the order of the parameters,
it means you changed the signature. You have two different methods.**

```
class Example {  
  
    public String add(String a, int b){  
        String c = a+b;  
        return c;  
    }  
    public int add(String y, int x){  
        int c = x;  
        return c;  
    }  
    public static void main(String args[]){  
    }  
}
```

```
class Example {  
  
    public String add(String a, int b){  
        String c = a+b;  
        return c;  
    }  
    public int add(int x, String y){  
        int c = x;  
        return c;  
    }  
    public static void main(String args[]){  
    }  
}
```



**3) If you change the name of the method,
it means you changed the signature. You have two different methods.**

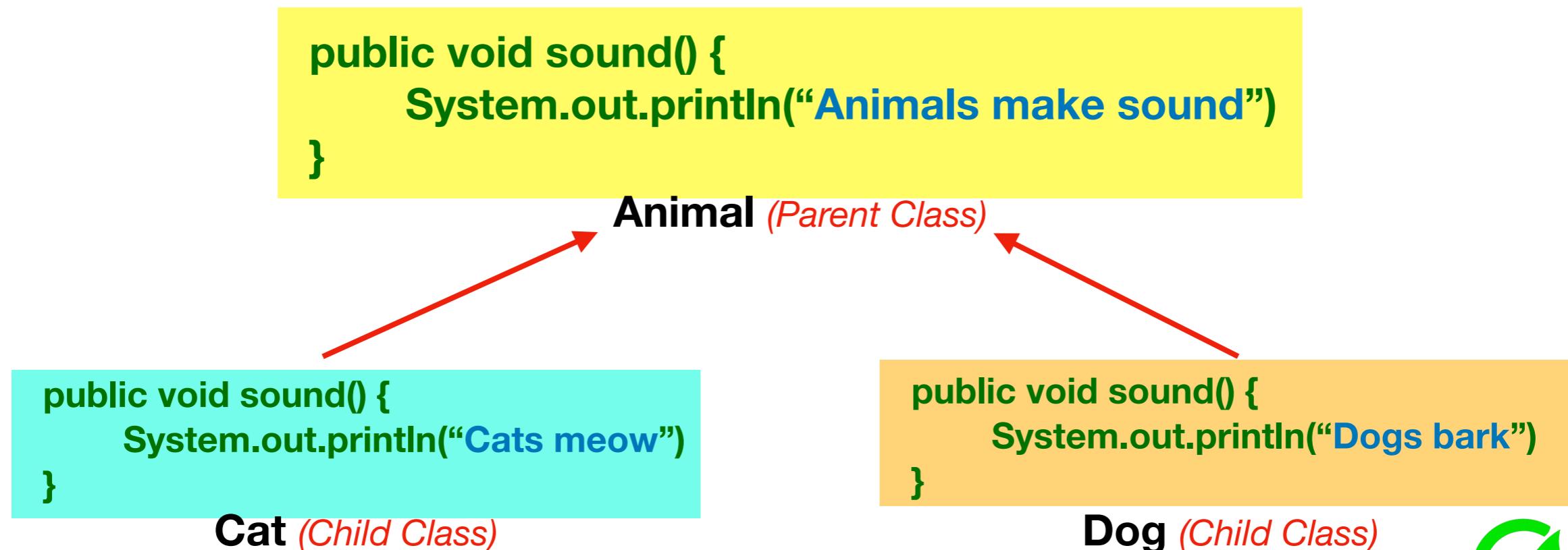
```
class Example {  
  
    public int add(int b){  
        int c = b;  
        return c;  
    }  
    private double add(int y){  
        int c = y;  
        return c;  
    }  
    public static void main(String args[]){  
  
    }  
}
```

```
class Example {  
  
    public int add(int b){  
        int c = b;  
        return c;  
    }  
    private double addChanged(int y){  
        int c = y;  
        return c;  
    }  
    public static void main(String args[]){  
  
    }  
}
```



What is Method Overriding ?

- 1) Declaring a method in child class which is **already present in parent class** is method overriding.
- 2) Overriding is done so that a child class can give its **own implementation** to a method which is already provided by the parent class.
- 3) In this case the method in parent class is called **overridden method** and the method in child class is called **overriding method**.



Overriding and Overridden Methods

```
public class Animal {  
    public void eat(){  
        System.out.println("Animals eat meat and vegetable");  
    }  
  
    public static void main(String[] args) {  
  
        Animal animal = new Animal();  
        animal.eat();  
  
    }  
}
```

→ Overriden Method

```
public class Dog extends Animal {  
    public void eat(){  
        System.out.println("Dogs eat meat");  
    }  
  
    public static void main(String[] args) {  
  
        Dog dog = new Dog();  
        dog.eat();  
  
    }  
}
```

```
public class Lamb extends Animal {  
    public void eat(){  
        System.out.println("Lambs eat grass");  
    }  
  
    public static void main(String[] args) {  
  
        Lamb lamb = new Lamb();  
        lamb.eat();  
  
    }  
}
```

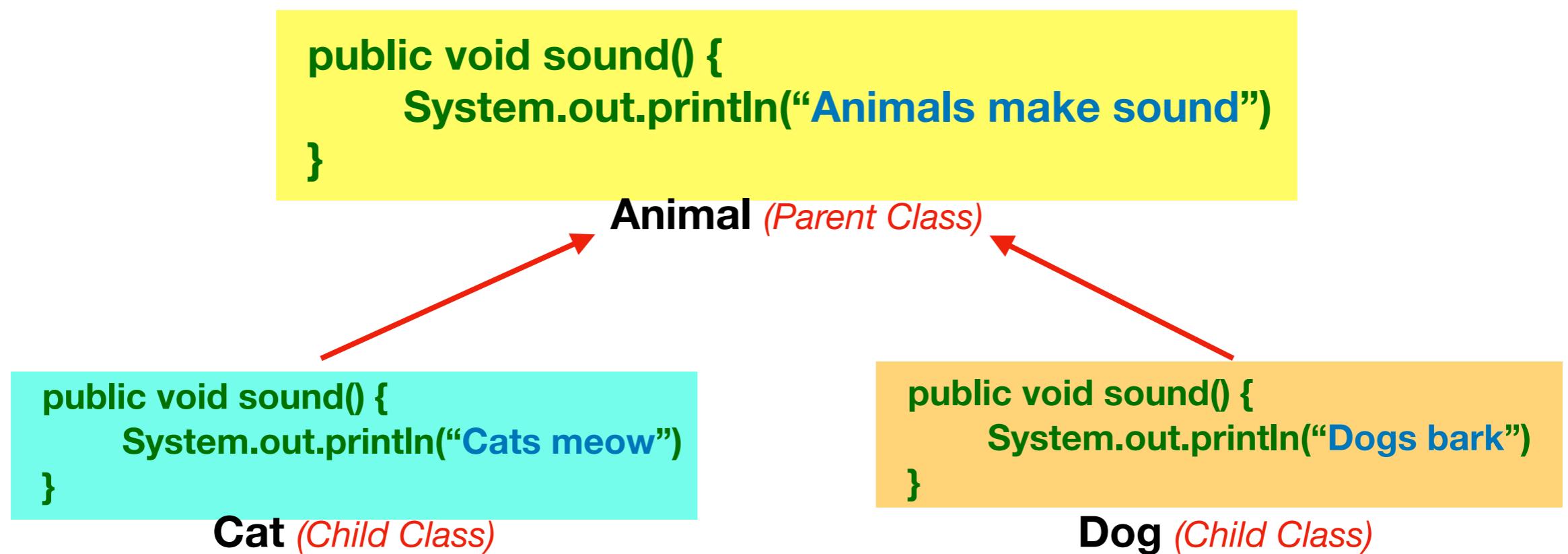
Overriding Method



Advantage of Method Overriding

The child class can give its own specific implementation to an inherited method without even modifying the parent class code.

As you can see in the image below, I used sound() method without modifying the parent class code



Access modifiers in “Overriding”

False

Child cannot restrict parent

```
public void add() {  
    System.out.println("Parent adding")  
}
```

Parent

```
protected void add() {  
    System.out.println("Child adding")  
}
```

Child extends Parent

True

Child's access modifier can be same with or larger than parent's access modifier

```
protected void add() {  
    System.out.println("Parent adding")  
}
```

Parent

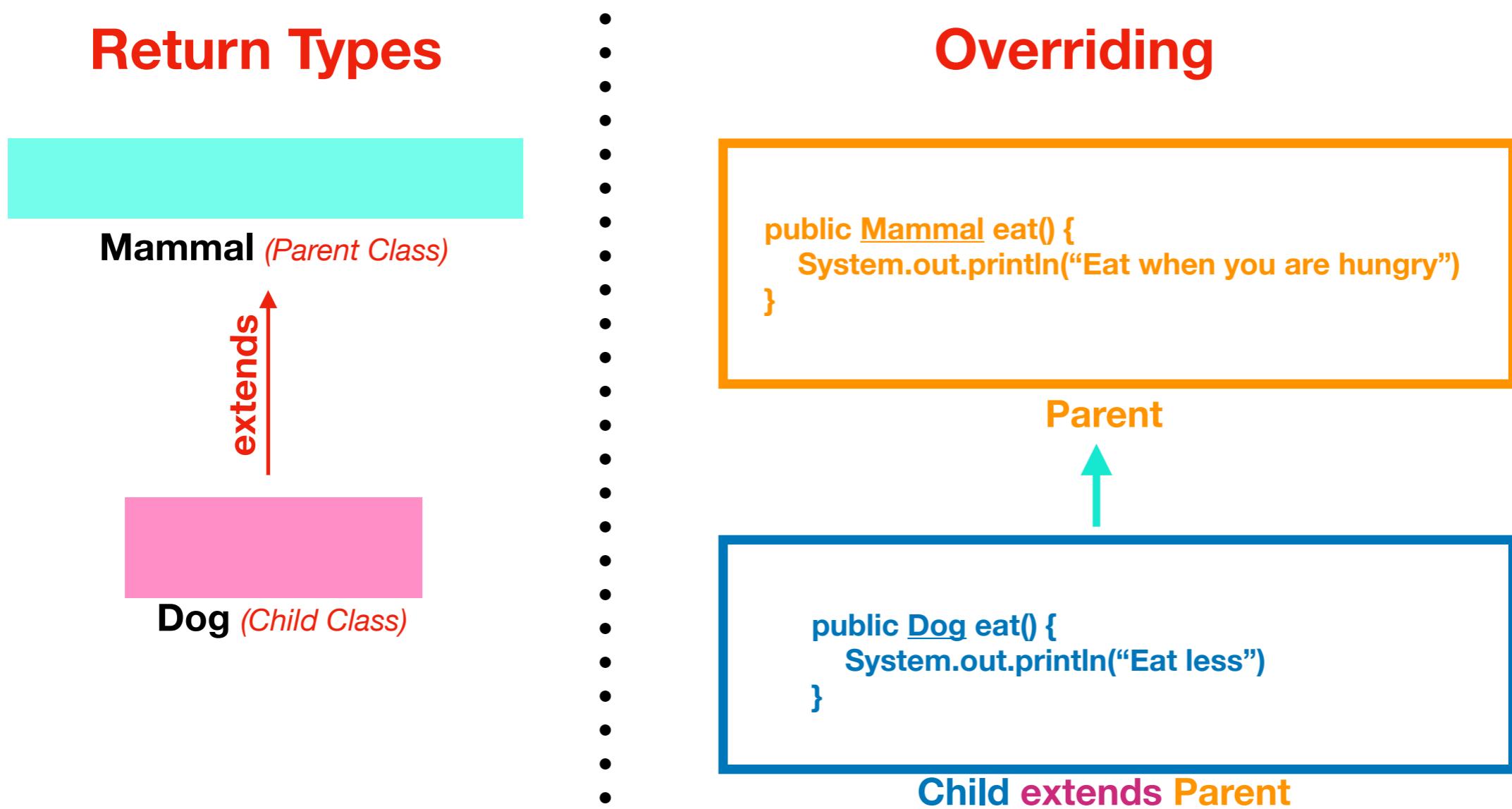
```
protected / public void add() {  
    System.out.println("Child adding")  
}
```



Return Types in “Overriding”

There must be IS-A relationship between the return types when you look at from overriding method(child) to overridden method(parent)

When you override a method, return type of overriding method (*method inside child*) must be **sub-class (child class) of return type of overridden method (*method inside parent*)**



Rules of Method Overriding

- 1) Do not change the signature (*name and parameters*) of the method**
- 2) Access modifier of the overriding method (*method of child class*) cannot be more restrictive than the overridden method (*method of parent class*) of parent class.**
- 3) Overriding method must use covariant return types**
- 4) private, static and final methods cannot be overridden.**
- 5) Overriding method (*method of child class*) can throw **compile time exceptions**, regardless of whether the overridden method (*method of parent class*) throws any exception or not.**
However, the overriding method should not throw **run time exceptions** that are new or broader than the ones declared by the overridden method.
- 6) If a class is extending an **abstract class** or implementing an **interface** then it has to override all the abstract methods unless the class itself is an abstract class.**



Review Questions

1) What is “method signature” ? Which methods are same according to Java ?

Method signature includes “method name” and “parameter list”

If the method signature is same for two methods, then java accepts them as same methods

2) What is polymorphism ?

Polymorphism includes “overloading” and “overriding”

3) What is the difference between “Overloading” and “Overriding” ?

Overloading is changing just parameters, overriding cannot touch the signature, it changes the body of the method

4) What is the advantage of “Overriding” ?

Multiple implementation, reusability



Note : We can use both overriden (method from parent class) and overriding (method from child class) methods in child class by using “super” keyword.

```
public class Lamb extends Animal {  
    public void eat(){  
        super.eat();  
        System.out.println("Lambs eat grass");  
    }  
  
    public static void main(String[] args) {  
        Lamb lamb = new Lamb();  
        lamb.eat();  
    }  
}
```



What is “Polymorphism” in Programming?

Polymorphism (*variety of forms*) allows us **define a method** and have its **multiple implementations**.

Polymorphism = Overloading + Overriding

```
public class Animal {  
    public void eat(){  
        System.out.println("Animals eat meat and vegetable");  
    }  
  
    public static void main(String[] args) {  
  
        Animal animal = new Animal();  
        animal.eat();  
    }  
}
```

```
public class Dog extends Animal {  
    public void eat(){  
        System.out.println("Dogs eat meat");  
    }  
  
    public static void main(String[] args) {  
  
        Dog dog = new Dog();  
        dog.eat();  
    }  
}
```

```
public class Lamb extends Animal {  
    public void eat(){  
        System.out.println("Lambs eat grass");  
    }  
  
    public static void main(String[] args) {  
  
        Lamb lamb = new Lamb();  
        lamb.eat();  
    }  
}
```



Types of “Polymorphism”

1) Method Overloading is a **compile time (static)** polymorphism.

This allows us to have **more than one method having the same name**, if the **parameters** of methods are **different** in number, sequence and data types of parameters.

2) Method Overriding is a **run time (dynamic)** polymorphism.

This allows us to have **more than one method having the same signature**, and the **body** of the methods **have different codes**.



Test Questions About Overriding

1)

```
class Derived {  
  
    public void getDetails(String temp){  
        System.out.println("Derived class " + temp);  
    }  
}  
  
public class Test extends Derived {  
  
    public int getDetails(String temp){  
        System.out.println("Test class " + temp);  
        return 0;  
    }  
  
    public static void main(String[] args){  
        Test obj = new Test();  
        obj.getDetails("GFG");  
    }  
}
```

The question is taken from Geeks for Geeks

- a) Derived class GFG
- b) Test class GFG
- c) Compilation error
- d) Runtime error



2)

```
class Derived{  
  
    public void getDetails(){  
        System.out.println("Derived class");  
    }  
}  
  
public class Test extends Derived{  
    protected void getDetails(){  
        System.out.println("Test class");  
    }  
  
    public static void main(String[] args){  
        Derived obj = new Test();  
        obj.getDetails();  
    }  
}
```

The question is taken from Geeks for Greek

- a) Test class
- b) Compilation error due to line xyz
- c) Derived class
- d) Compilation error due to access modifier



3) class Derived{

```
    public void getDetails(){
        System.out.printf("Derived class ");
    }
}
```

```
public class Test extends Derived{
    public void getDetails(){
        System.out.printf("Test class ");
        super.getDetails();
    }

    public static void main(String[] args){
        Derived obj = new Test();
        obj.getDetails();
    }
}
```

The question is taken from Geeks for Geeks

- a) Test class Derived class
- b) Derived class Test class
- c) Compilation error
- d) Runtime error



4)

```
class Derived
{
    protected final void getDetails()
    {
        System.out.println("Derived class");
    }
}

public class Test extends Derived
{
    protected final void getDetails()
    {
        System.out.println("Test class");
    }
    public static void main(String[] args)
    {
        Derived obj = new Derived();
        obj.getDetails();
    }
}
```

- a) Derived class
- b) Test class
- c) Runtime error
- d) Compilation error

The question is taken from Geeks for Geek



5)

What will be the output of the following program?

```
class Person {  
    public void talk() {  
        System.out.print("First Program");  
    }  
}  
class Student extends Person {  
    public void talk() {  
        System.out.print("Second Program");  
    }  
}  
public class TestProgram {  
    public static void main(String args[]) {  
        Person p = new Student();  
        p.talk();  
    }  
}
```

The question is taken from Merit Campus



6)

What will be the output of the following program? Assume that all the classes belong to the same package.

```
class CURD {  
  
    public static void main(String[] args) {  
        new C().create();  
        new D().update();  
        new R().read();  
        new D().delete();  
    }  
}  
  
class C {  
    public void create() { System.out.print("c"); }  
}  
  
class U {  
    private void update() { System.out.print("u"); }  
}  
  
class R extends C {  
    public void create() { System.out.print("C"); }  
    protected void read() { System.out.print("R"); }  
}  
  
class D extends U {  
    void update() { System.out.print("U"); }  
    void delete() { System.out.print("D"); }  
}
```

The question is taken from Merit Campus



7)

What will be the output of the following program?

```
class Super {  
    public Integer getLength() {  
        return new Integer(4);  
    }  
}  
public class Sub extends Super {  
    public Long getLength() {  
        return new Long(5);  
    }  
    public static void main(String[] args) {  
        Super sooper = new Super();  
        Sub sub = new Sub();  
        System.out.println(sooper.getLength().toString() + ", " + sub.getLength().toString());  
    }  
}
```

The question is taken from Merit Campus



8)

What will be the output of the following program?

```
public class MethodOverriding {
    public static void main(String args[]) {
        X x = new X();
        Y y = new Y();
        y.m2();
        x.m1();
        y.m1();
        x = y;
        x.m1();
    }
}
class X {
    public void m1() {
        System.out.println("m1 ~ X");
    }
}
class Y extends X {
    public void m1() {
        System.out.println("m1 ~ Y");
    }
    public void m2() {
        System.out.println("m2 ~ Y");
    }
}
```

The question is taken from Merit Campus



9)

What will be the output of the following program?

```
public class Outer {
    public static void main(String args[]) {
        Computer mouse = new Laptop();
        System.out.println(mouse.getValue(100, 200));
    }
}
class NoteBook {
    int getValue(int a, int b) {
        if (a > b)
            return a;
        else
            return b;
    }
}
class Computer extends NoteBook {
    int getValue(int a, int b) {
        return a * b;
    }
}
class Laptop extends Computer {
    int getValue(int a, int b) {
        return b - a;
    }
}
```

The question is taken from Merit Campus



10)

What will be the output of the following program?

```
public class Product {
    public static void main(String[] args) {
        M m = new M();      M n = new N();
        M o = new O();      O oo = new O();
        m.product(3);      n.product(3);
        oo.product(3);
    }
}
class M {
    int product(int i) {
        int result = i * i;
        System.out.print("{" + i + ", " + result + "}~");
        return result;
    }
}
class N extends M {
    int product(int i) {
        int result = i + i;
        System.out.print("[{" + i + ", " + result + "}]~");
        return result;
    }
}
class O extends M {
    int product(int i) {
        int result = i * 2;
        System.out.print("(" + i + ", " + result + ")~");
        return result;
    }
}
```

The question is taken from Merit Campus



What is Exception ?



Everything is good, no problem. No exception



The man is handling exception



The man is throwing exception

When executing Java code, different errors can occur:

- 1) **Coding errors made by the programmer,** (*Car is not secure enough*) - (*Trying to access an invalid index in an array.*)
- 2) **Errors due to wrong input,** (*Visitor left the car*) - (*Dividing a number by zero*)
- 3) **Unforeseeable things.** (*Wild animals attacked the car*) - (*Internet connection is down*)

When an error occurs, Java will normally stop (*End the safari*) **and generate an error message** (*SOS*).

The technical term for this is Java will “**throw an exception**”



Exception Types

1) **Checked Exceptions** (*Compile time exceptions*): Checked exceptions are checked at compile-time.

```
class Example {  
    public static void main(String args[]){  
        FileInputStream fis = new FileInputStream( name: "B:/myfile.txt");  
        int k;  
  
        while(( k = fis.read() ) != -1){  
            System.out.print((char)k);  
        }  
  
        fis.close();  
    }  
}
```

Note 1: They must be handled or declared.

Common Compile Time Exceptions

- 1) **FileNotFoundException**: Thrown when code tries to reference a file that does not exist
- 2) **IOException**: (IO stands for InputOutput) Thrown when there's a problem reading or writing a file

Note 2: Keep in mind that **FileNotFoundException** is a subclass of **IOException**



How to handle “Checked Exceptions” ?

1) Declare the exception using “throws” keyword.

```
class Example {  
    public static void main(String args[]) throws IOException {  
        FileInputStream fis = new FileInputStream( name: "B:/myfile.txt");  
        int k;  
  
        while(( k = fis.read() ) != -1)  
        {  
            System.out.print((char)k);  
        }  
        fis.close();  
    }  
}
```

“try-catch blocks” is better, because
1) It is more readable
2) It gives message to user after running

2) Handle by using “try-catch blocks”

```
class Example {  
    public static void main(String args[]) {  
  
        FileInputStream file = null;  
  
        try{  
            file = new FileInputStream( name: "B:/myfile.txt");  
        } catch(IOException e) {  
            System.out.println("The specified file is not present at the given path");  
        }  
    }  
}
```



Try Catch Blocks

- 1) The “try block” contains set of statements where an exception can occur.
- 2) A try block is always followed by a “catch block”, which handles the exception that occurs in associated try block.
- 3) A try block must be followed by **catch blocks** or **finally block** or both.

```
class Example {
    public static void main(String args[]) {
        FileInputStream file = null;
        try{
            file = new FileInputStream( name: "B:/myfile.txt");
        } catch(IOException e) {
            System.out.println("The specified file is not present at the given path");
        }
    }
}
```

```
class Example {
    public static void main(String args[]) {
        FileInputStream file = null;
        try{
            file = new FileInputStream( name: "B:/myfile.txt");
        } catch(IOException e) {
            System.out.println("The specified file is not present at the given path");
        }
    }
}
```

If there is no exception
catch block does not work

finally block runs whether exception
occurs or not. Like closing connection...



4) While writing a program, if you think that certain statements in a program can throw an exception, enclosed them in try block and handle that exception

```
class Example {  
    public static void main(String args[]) {  
  
        FileInputStream file = null;  
  
        try{  
  
            file = new FileInputStream( name: "B:/myfile.txt");  
        } catch(IOException e) {  
  
            System.out.println("The specified file is not present at the given path");  
        }  
    }  
}
```

*When you want to get a file,
maybe the path will be wrong.
Therefore, you need to put it inside the
try block.*



5) A single try block can have several catch blocks associated with it. You can catch different exceptions in different catch blocks.

```
class Example {  
    public static void main(String args[]) {  
        int num1, num2;  
        try {  
            /* We suspect that this block of statement can throw  
             * exception so we handled it by placing these statements  
             * inside try and handled the exception in catch block  
             */  
            num1 = 0;  
            num2 = 62 / num1;  
            System.out.println(num2);  
            System.out.println("Hey I'm at the end of try block");  
        }  
        catch (ArithmaticException e) {  
            /* This block will only execute if any Arithmatic exception  
             * occurs in try block  
             */  
            System.out.println("You should not divide a number by zero");  
        }  
        catch (Exception e) {  
            /* This is a generic Exception handler which means it can handle  
             * all the exceptions. This will execute if the exception is not  
             * handled by previous catch blocks.  
             */  
            System.out.println("Exception occurred");  
        }  
        System.out.println("I'm out of try-catch block in Java.");  
    }  
}
```

- 1) If exception is caught in first catch block, second one does not run.
- 2) Second exception must be parent class of the first exception.



What is output ?

```
String s = "";
try {
    s += "t";
} catch(Exception e) {
    s += "c";
} finally {
    s += "f";
}
s += "a";
System.out.print(s);
```



Review Questions

True or False Questions

- 1) try block must be used with catch block or/and finally block**
- 2) finally block gets executed always**
- 3) We can have multiple catch blocks with a try block**
- 4) If you use multiple catch blocks, child exception must be the first**
- 5) Explain, what the “[FileNotFoundException](#)” is.**
- 6) Explain, what the “[IOException](#)” is.**



Review Questions Answers

True or False Questions

- 1) try block must be used with catch block ==> False
- 2) finally block gets executed always ==> True
- 3) We can have multiple catch blocks with a try block ==> True
- 4) If you use multiple catch blocks, child exception must be the first ==> True

Question

5) Explain, what the “[FileNotFoundException](#)” is.

When your application searching to read a file, if the **file is not found** then there is a [FileNotFoundException](#) to be thrown.

[FileNotFoundException](#) is a [subclass](#) of [IOException](#).

6) Explain, what the “[IOException](#)” is.

[IOExceptions](#) are thrown when there is any input / output file operation issues while application performing certain tasks accessing the files.

[IOException](#) is a [checked exception](#)(Compile Time Exception) and application developer has to handle in correct way.



IOException in Java

THIS CLASS IS THE GENERAL CLASS OF EXCEPTIONS PRODUCED BY FAILED OR INTERRUPTED I/O OPERATIONS.

AS THIS IS A CHECKED EXCEPTION, IT MUST BE HANDLED BY THE PROGRAMMER ELSE PROGRAM DOES NOT COMPILE

BELOW ARE SOME SCENARIOS WHEN IOException WOULD BE THROWN :

- ✓ YOU WERE READING NETWORK FILE AND GOT DISCONNECTED.
- ✓ READING LOCAL FILE WHICH IS NOT AVAILABLE ANY MORE.
- ✓ USING SOME STREAM TO READ THE DATA AND SOME OTHER PROCESS CLOSES THE STREAM.
- ✓ YOU ARE TRYING TO READ/WRITE A FILE AND DON'T HAVE PERMISSION
- ✓ YOU WERE WRITING A FILE AND DISK SPACE IS NOT AVAILABLE ANYMORE



What is output ?

```
public static void add(){
    String str = "";
    System.out.println(str.length());
}
```

What is output ?

```
public static void add(){
    String str = null;
    System.out.println(str.length());
}
```



What is output ?

```
public class Test4 {  
  
    public static void main(String[] args) {  
        System.out.println(exceptions());  
    }  
    @SuppressWarnings("finally")  
    public static String exceptions() {  
        String result = "";  
        String v = null;  
        try {  
            try {  
                result = result + "a";  
                v.length();  
                result = result + "b";  
            }catch(NullPointerException e){  
                result = result + "c";  
            }finally {  
                result = result + "d";  
                throw new Exception();  
            }  
        }catch(Exception e) {  
            result = result + "e";  
        }  
        return result;  
    }  
}
```

acde



How to print exception on the console

```
1) class Example {  
    public static void main(String args[]) {  
        try {  
            hop();  
        } catch (Exception e) {  
            System.out.println(e);  
        }  
    }  
  
    private static void hop() {  
        throw new RuntimeException("cannot hop");  
    }  
}
```



2)

```
class Example {  
  
    public static void main(String args[]) {  
  
        try {  
            hop();  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
  
    private static void hop() {  
        throw new RuntimeException("cannot hop");  
    }  
}
```



3) **class** Example {

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

```
        try {
```

```
            hop();
```

```
        } catch (Exception e) {
```

```
            e.printStackTrace();
```

```
        }
```

```
}
```

```
    private static void hop() {
```

```
        throw new RuntimeException("cannot hop");
```

```
}
```

```
}
```



2) Unchecked Exceptions (*Run time exceptions*): Unchecked exceptions are not checked at compile time. You will get the exceptions after running the code.

```
class Example {  
    public static void main(String args[])  
    {  
        int num1=10;  
        int num2=0;  
  
        int res=num1/num2;  
        System.out.println(res);  
    }  
}
```



Note 1: They don't have to be handled or declared.

Common Run Time Exceptions:

1) ArithmeticException: Thrown when code attempts to divide by zero

```
System.out.println(12 / 0);
```

2) ArrayIndexOutOfBoundsException: Thrown when code uses an illegal index to access an array

```
int arr[] = {a, b, c} System.out.println( arr[12] );
```

3) ClassCastException: Thrown when code tries to cast inappropriate data types

```
Object obj = 70;
```

```
String s = (String) obj;
```



4) **IllegalArgumentException:**

```
public static void setNumberEggs(int numberEggs) {  
    if (numberEggs < 0){  
        throw new IllegalArgumentException( "# eggs must not be negative");  
    }  
    this.numberEggs = numberEggs;  
}
```

5) **NullPointerException:** Thrown **when** there is a **null** reference where an **object** is required

```
String name = null;  
public void printLength(){  
    System.out.println(name.length());  
}
```

6) **NumberFormatException:** Thrown **when** an attempt is made to **convert a string to a numeric type** but the string **doesn't have an appropriate format**

```
int num = Integer.parseInt("abc");
```



7) **StringIndexOutOfBoundsException:**

```
String str = "Hello World";
char charAtNegativeIndex = str.charAt(-1); // Trying to access at negative index
char charAtLengthIndex = str.charAt(11); // Trying to access at index equal to size of the string
```

8) **IllegalStateException:** IllegalStateException signals that a method's been invoked at an illegal or inappropriate time.

```
Iterator<Integer> intListIterator = new ArrayList<>().iterator(); //Initialized with index at -1
intListIterator.remove(); // IllegalStateException
```



Difference between throw and throws

```
void Demo() throws ArithmeticException, NullPointerException{  
    throw new ArithmeticException();  
}
```

THROW	THROWS
throw keyword is used to throw an exception explicitly.	throws keyword is used to declare one or more exceptions, separated by commas.
Only single exception is thrown by using throw.	Multiple exceptions can be thrown by using throws.
throw keyword is used within the method.	throws keyword is used with the method signature.
Syntax wise throw keyword is followed by the instance variable.	Syntax wise throws keyword is followed by exception class names.



How to create Custom Exception in Java

Java exceptions cover almost all general exceptions that are bound to happen in programming. However, we sometimes need to supplement these standard exceptions with our own.

Checked Custom Exception

```
public class IncorrectFileNameException extends Exception {  
    public IncorrectFileNameException(String errorMessage) {  
        super(errorMessage);  
    }  
}
```

Unchecked Custom Exception

```
public class IncorrectFileExtensionException extends RuntimeException {  
    public IncorrectFileExtensionException(String errorMessage, Throwable err) {  
        super(errorMessage, err);  
    }  
}
```

Custom exceptions are very useful when we need to handle specific exceptions related to the business logic.



1)

Which of the following pairs fill in the blanks to make this code compile? (Choose all that apply)

```
7: public void ohNo() _____ Exception {  
8:     _____ Exception();  
9: }
```

- A. On line 7, fill in throw
- B. On line 7, fill in throws
- C. On line 8, fill in throw
- D. On line 8, fill in throw new
- E. On line 8, fill in throws
- F. On line 8, fill in throws new



2)

Which exception will the following throw?

```
Object obj = new Integer(3);  
String str = (String) obj;  
System.out.println(str);
```

- A. `ArrayIndexOutOfBoundsException`
- B. `ClassCastException`
- C. `IllegalArgumentException`
- D. `NumberFormatException`
- E. None of the above.



3)

What will happen if you add the statement `System.out.println(5 / 0);` to a working `main()` method?

- A.** It will not compile.
- B.** It will not run.
- C.** It will run and throw an `ArithmaticException`.
- D.** It will run and throw an `IllegalArgumentException`.
- E.** None of the above.



4)

Which of the following can be inserted in the blank to make the code compile? (Choose all that apply)

```
public static void main(String[] args) {  
    try {  
        System.out.println("work real hard");  
    } catch (_____ e) {  
    } catch (RuntimeException e) {  
    }  
}
```

A. Exception
B. IOException
C. IllegalArgumentException
D. RuntimeException



5)

What is the output of the following snippet, assuming a and b are both 0?

```
3:     try {  
4:         return a / b;  
5:     } catch (RuntimeException e) {  
6:         return -1;  
7:     } catch (ArithmetricException e) {  
8:         return 0;  
9:     } finally {  
10:        System.out.print("done");  
11:    }
```

- A.** -1
- B.** 0
- C.** done-1
- D.** done0
- E.** The code does not compile.
- F.** An uncaught exception is thrown.



6) What is the output of the following program?

```
1: public class Dog {  
2:     public String name;  
3:     public void parseName() {  
4:         System.out.print("1");  
5:         try {  
6:             System.out.print("2");  
7:             int x = Integer.parseInt(name);  
8:             System.out.print("3");  
9:         } catch (NumberFormatException e) {  
10:             System.out.print("4");  
11:         }  
12:     }  
13:     public static void main(String[] args) {  
14:         Dog leroy = new Dog();  
15:         leroy.name = "Leroy";  
16:         leroy.parseName();  
17:         System.out.print("5");  
18:     } }
```

- A.** 12
- B.** 1234
- C.** 1235
- D.** 124
- E.** 1245
- F.** The code does not compile.
- G.** An uncaught exception is thrown.



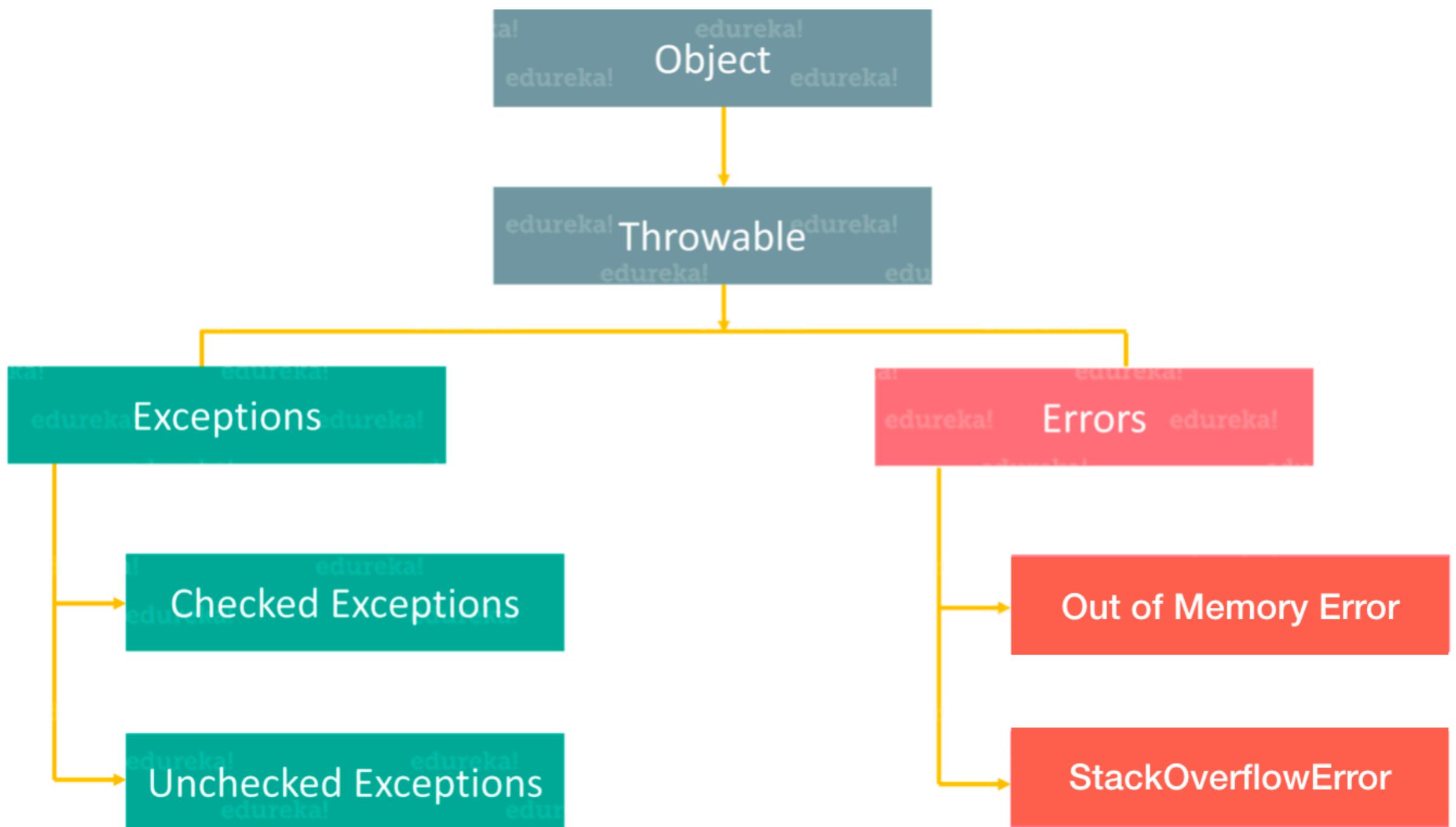
Answers

- 1)** B, D. In a method declaration, the keyword throws is used. To actually throw an exception, the keyword throw is used and a new exception is created.
- 2)** B. The second line tries to cast an Integer to a String. Since String does not extend Integer, this is not allowed and a ClassCastException is thrown.
- 3)** C. The compiler tests the operation for a valid type but not a valid result, so the code will still compile and run. At runtime, evaluation of the parameter takes place before passing it to the print() method, so an ArithmeticException object is raised.



- 4)** C, E. Option C is allowed because it is a more specific type than `RuntimeException`. Option E is allowed because it isn't in the same inheritance tree as `RuntimeException`. It's not a good idea to catch either of these. Option B is not allowed because the method called inside the try block doesn't declare an `IOException` to be thrown. The compiler realizes that `IOException` would be an unreachable catch block. Option D is not allowed because the same exception can't be specified in two different catch blocks. Finally, option A is not allowed because it's more general than `RuntimeException` and would make that block unreachable.
- 5)** E. The order of catch blocks is important because they're checked in the order they appear after the try block. Because `ArithmeticException` is a child class of `RuntimeException`, the catch block on line 7 is unreachable. (If an `ArithmeticException` is thrown in try try block, it will be caught on line 5.) Line 7 generates a compiler error because it is unreachable code.
- 6)** E. The `parseName` method is invoked within `main()` on a new `Dog` object. Line 4 prints 1. The try block executes and 2 is printed. Line 7 throws a `NumberFormatException`, so line 8 doesn't execute. The exception is caught on line 9, and line 10 prints 4. Because the exception is handled, execution resumes normally. `parseName` runs to completion, and line 17 executes, printing 5. That's the end of the program, so the output is 1245.





Errors

- 1) Errors are the conditions which **cannot get recovered** by any handling techniques.
- 2) It surely **cause termination** of the program abnormally.
- 3) Errors **belong to unchecked** type and mostly **occur at runtime**.
- 4) Some of the examples of errors are **Out of Memory Error** or a **System Crash Error**.

```
class Example {  
  
    public static void main(String args[]) {  
  
        for(int i=0; i<3; i--){  
            System.out.println(i);  
        }  
    }  
}
```



finally keyword is used with try-catch block to provide statements that will always gets executed even if some exception arises, usually **finally** is used to close resources.

finalize() method is executed by Garbage Collector before the object is destroyed.



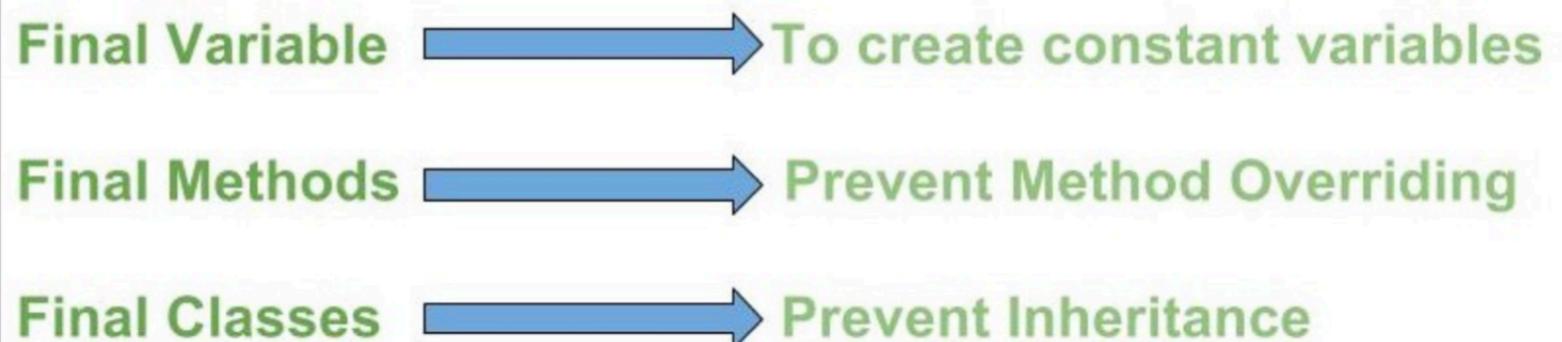
Garbage



Finalized Garbage



Object Destroyed



Interview Question

What is difference between **final**, **finally** and **finalize** keywords in Java?

Answer

1) **final** and **finally** are keywords in java whereas **finalize** is a method.

2) **final keyword** can be used;

with **class variables** so that they **can't be reassigned**,

final double pi = 3.14; ==> pi = 3.1412; X

final double pi; ==> pi = 3.14; ✓ ==> pi = 3.1412; X

with **class** to avoid extending by classes,

public class Animal{ } ==> public class Dog extends Animal { } ✓

public final class Animal{ } ==> public class Dog extends Animal { } X

with **methods** to avoid overriding by subclasses,

public void eat() { System.out.println("Overridden method") { } ==> public void eat() { System.out.println("Overriding method") } ✓

public final void eat() { System.out.println("Overridden method") { } ==> public void eat() { System.out.println("Overriding method") } X



Review Questions

True / False Questions

- 1) A Java class that extends another class inherits all of its **public** and **protected** methods and variables.**
- 2) `this()` calls in class constructor, `super()` calls parent constructor.**
- 3) `this` and `super` must be the first statement in a constructor.**
- 4) Overriding and Overridden methods may have the different signature.**
- 5) Child method at least as accessible as the parent method.**
- 6) Child method's return type must be sub-class of parent method's return type.**
- 7) Child method must not declare any new or broader exceptions.**
- 8) When the method signature is different, with the method taking different inputs, it is referred to as method overloading.**

public Animal feed(){}

public Animal feed(){}



public Cat feed(){}



public Creature feed(){}



Abstract Class

To create an abstract class, use “**abstract**” keyword between the “**access modifier**” and the “**class**” keyword

```
public abstract class Animal() { }
```

Note: An abstract class **cannot be instantiated**, which means you are **not allowed to create an object** of it.

public abstract class Animal() { } ==> Animal animal = new Animal(); X

Abstract Method

Note: A **method without body** (*no implementation*) is known as **abstract method**.

Concrete Method

```
public int sumOfTwo(int n1, int n2) { return n1 + n2; }
```

Abstract Method

```
public abstract int sumOfTwo(int n1, int n2);
```



Do Not Put Body for the Abstract Methods

```
public abstract int sumOfTwo(int n1, int n2) { } // You get Compile Time Error
```



Do Not Forget to Put Body for the Concrete Methods

```
public int sumOfTwo(int n1, int n2); // You get Compile Time Error
```



Note: An **abstract method** must always be declared **in an abstract class**, or in other words; if a class has an abstract method, then the class should be declared abstract as well.

```
public class Animal {  
    public abstract void sound();  
}
```



```
public abstract class Animal {  
    public abstract void sound();  
}
```



Note: An **abstract class** may have both **abstract methods** and **concrete methods**.

```
public abstract class Animal {  
    public abstract void sound();  
}
```



```
public abstract class Animal {  
    public void sound(){ }  
}
```



```
public abstract class Animal {  
    public abstract void eat();  
    public void sound(){ }  
}
```



Note: If a regular class extends an abstract class, then the class must implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

```
public abstract class Animal {  
    public abstract void sound();  
}
```

Regular child classes must override abstract methods

```
public class Dog extends Animal {  
    public void sound(){  
        System.out.println("Barking");  
    }  
}
```

```
public abstract class Animal {  
    public abstract void sound();  
}
```

If child class is abstract as well
no need to override abstract methods

```
public abstract class Dog extends Animal {  
}
```



Is there any compile time error ?

```
public abstract class Animal {  
    public abstract String getName();  
}
```

```
public abstract class BigCat extends Animal {  
    public abstract void roar();  
}
```

```
public class Lion extends BigCat {  
    public String getName() {  
        return "Lion";  
    }  
    public void roar() {  
        System.out.println("The Lion lets out a loud ROAR!");  
    }  
}
```



Is there any compile time error ?

```
public abstract class Animal {  
    public abstract String getName();  
}
```

```
public abstract class BigCat extends Animal {  
    public String getName() {  
        return "BigCat";  
    }  
    public abstract void roar();  
}
```

```
public class Lion extends BigCat {  
    public void roar() {  
        System.out.println("The Lion lets out a loud ROAR!");  
    }  
}
```



Why do we need “abstraction” ?

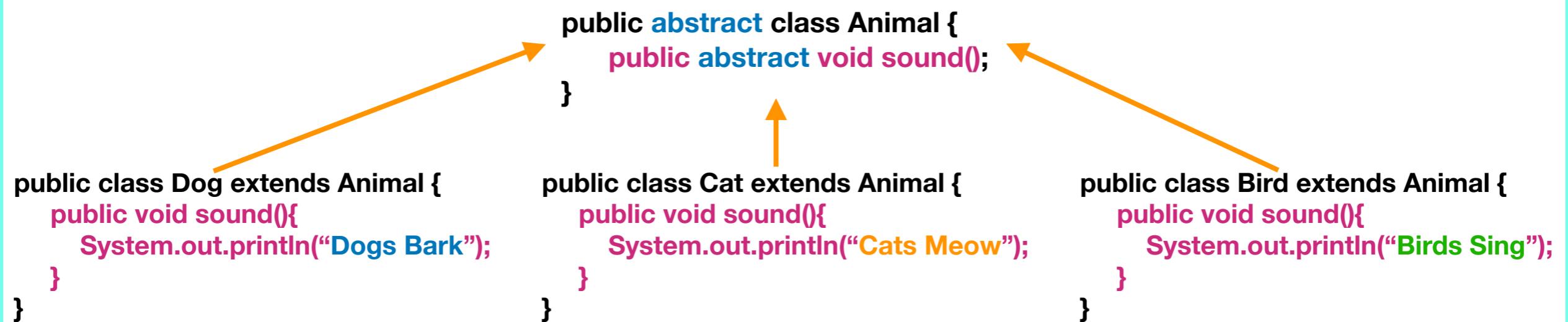
What kind of sound does an “Animal” make?

An “Animal” makes a sound but we cannot define a specific sound for an “Animal”. So, there is sound but we cannot make a specific definition.

To make sure every child class has some functionalities?

For example, we are making a research about the sounds of the animals; therefore, we need to define sound() method for every child class. To make sure every child class of Animal class has sound() method, we make it abstract.

As you can see in the following examples, we can make specific definitions for the sounds of Dog, Cat, etc; however, we cannot make a specific definition for the sound of an Animal.
Therefore, we do not need to create body for the method, we just declare that Animals make sound.



Can we make an abstract class “final” ?

```
public final abstract class Animal {  
    public abstract void sound();  
}
```



```
public class Dog extends Animal{  
    public void sound(){  
        System.out.println("Dogs Bark");  
    }  
}
```

An **abstract class** is one that **must be extended by another class** to be instantiated.
However, a final class can't be extended by another class.
So the compiler refuses to process the code and we get **Compile Time Error**



Can we make an abstract method “final” ?

```
public abstract class Animal {  
    public final abstract void sound();  
}
```



```
public class Dog extends Animal{  
    public void sound(){  
        System.out.println("Dogs Bark");  
    }  
}
```

An **abstract method** is one that **must be overridden**.
Once marked as final, the method cannot be overridden in a subclass.



Can we make an abstract method “private” ?

```
public abstract class Animal {  
    private abstract void sound();  
}
```



```
public class Dog extends Animal{  
    sound() method cannot be accessed from  
    child class because it is private  
}
```

An **abstract method** is one that **must be overridden**.
however, if the method is **private**, it **cannot be overridden in a child class because it cannot be accessed**.



Abstract Class Rules:

- 1. We cannot create objects from Abstract Classes.**
- 2. Abstract classes may have or may not have abstract and non-abstract methods**
- 3. Abstract classes cannot be private or final.**
- 4. An abstract class that extends another abstract class inherits all of its abstract methods automatically. Therefore, no need to override.**
- 5. The first concrete class that extends an abstract class must provide an implementation for all of the inherited abstract methods.**



Abstract Method Definition Rules:

- 1. Abstract methods **may only be defined in abstract classes.****
- 2. Abstract methods **may not be declared private or final or static.****
- 3. Abstract methods **must not provide a method body in the abstract class.****
- 4. Implementing an abstract method in a subclass follows the same rules for overriding a method.**

For example, the name and signature must be the same, and the visibility of the method in the subclass must be at least as accessible as the method in the parent class.



Interface

Interface looks like a class but it is not a class.

An interface can have just abstract methods.

Variables declared in an interface are public, static , and final. Marking a variable as private or protected will trigger a compiler error

You have to initialize the variable when you declare, otherwise you get Compile Time Error

```
interface Example {  
    public void add(); // methods must be abstract  
    public static final int num1 = 12; // variables are public static and final  
}
```

```
interface Example {  
    void add(); // methods are public by default  
    int num1 = 12; // Variables are public static and final by default  
}
```

```
interface Example {  
    int num1; // Compile Time Error  
    int num1 = 12; // It is okay because it is initialized  
}
```

Note: public void add(); and void add(); are same for interfaces

Note: public static final int num1 = 12; and int num1 = 12; are same for interfaces



Why do we need “interface” in Java ?

- 1: **Concrete class that implements interface must implement all the methods of the interface.**
- 2: **Java programming language does not allow you to extend more than one class,**
However, you can implement more than one interfaces in your class.

```
interface Animal {  
    void eat () ;  
    void drink () ;  
}  
  
interface Mammal {  
    void nurse () ;  
    void blood () ;  
    void heart () ;  
}  
  
public class Cat implements Animal, Mammal {  
    void eat () {  
        System.out.println("Cats eat");  
    }  
    void drink () {  
        System.out.println("Cats drink");  
    }  
    void nurse () {  
        System.out.println("Cats nurse their young with milk");  
    }  
  
    void blood () {  
        System.out.println("Cats are warm-blooded");  
    }  
  
    void heart () {  
        System.out.println("Cats possess four-chambered hearts");  
    }  
}
```

```
interface Animal {  
    void eat () ;  
    void drink () ;  
}  
  
interface Mammal {  
    void nurse () ;  
    void blood () ;  
    void heart () ;  
}  
  
Public abstract class Cat implements Animal, Mammal {  
}
```

Note: Abstract class that implements interface
may not implement all the methods of the interface.



Key Points

1: You cannot create an object from an Interface.

A) It is abstract

B) It is not completed, because there are abstract methods inside it

2: All methods must be abstract inside the interface.

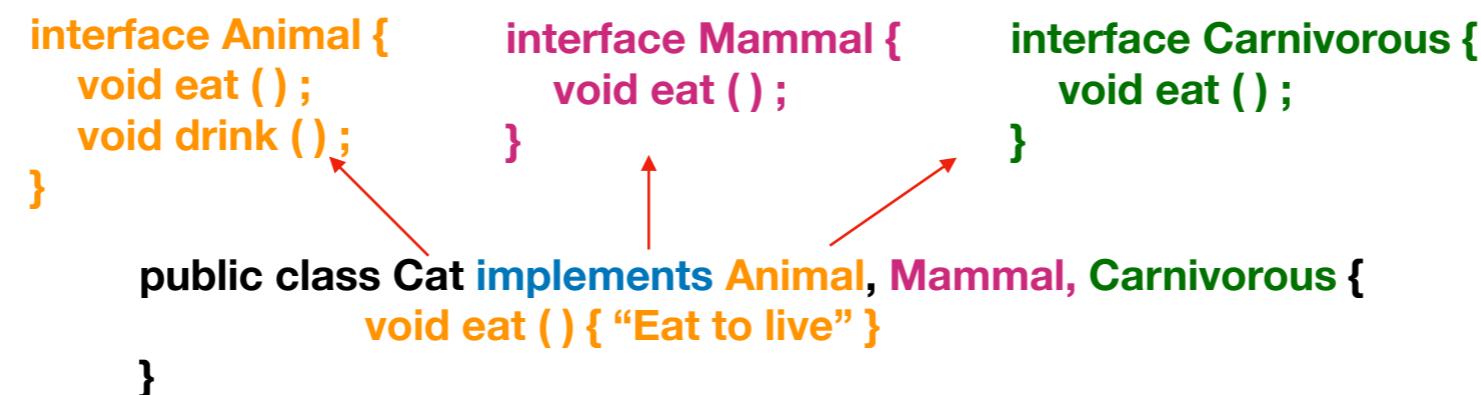
However, abstract classes can have both abstract and concrete methods

**3: Use “implements” keyword by classes to implement an interface, not extends.
But use “extends” keyword by interfaces to implement an interface.**

Class ==> interface : implements
Class ==> Class : extends
Interface ==> interface : extends

4: A class can implement any number of interfaces.

5: If there are two or more same methods in two interfaces and a class implements all interfaces, implementation of the method once is enough.



5: A class cannot implement more than one interfaces that have methods with same name but different return type.

```
interface Animal {  
    void eat () ;  
    void drink () ;  
}  
  
interface Mammal {  
    int eat () ;  
}  
  
interface Carnivorous {  
    String eat () ;  
}  
  
public class Cat implements Animal, Mammal, Carnivorous { // Compile Time Error  
    void eat () { “Eat to live” }  
}
```

6: Variable names conflicts can be resolved by interface name.

```
interface Animal {  
    int height = 10;  
}  
  
interface Mammal {  
    int height = 12 ;  
}  
  
interface Carnivorous {  
    int height = 14 ;  
}  
  
public class Cat implements Animal, Mammal, Carnivorous {  
    System.out.println(Animal.height); // ==> 10  
    System.out.println(Mammal.height); // ==> 12  
    System.out.println(Carnivorous.height); // ==> 14  
}
```



Warm Up 29

True / False Questions (*If there is any Compile Time Error, they are accepted as false*)

- T** 1) To create an object by using “**abstract class**” is not allowed.
- F** 2) To create an object by using “**interface**” is allowed.
- F** 3) `public abstract int sumOfTwo(int n1, int n2) { }`
- F** 4) `public class Animal {
 public abstract void sound();
}`
- T** 5) `public abstract class Animal {
 public abstract void eat();
 public void sound(){ }
}`
- T** 6) An abstract class cannot be final. (*According to the answer, type the reason*)
- F** 7) `public abstract class Animal {
 private abstract void sound();
}` (*According to the answer, type the reason*)
- T** 8) Java programming language **does not allow** you to extend more than one class
- T** 9) Java programming language **allows** you to implement more than one interface
- F** 10) An interface can have **abstract and concrete methods**.



7: If you need a ‘method with body’ (Concrete Method) in an interface, you must use “default” or “static” keyword explicitly

```
public interface Cat {  
  
    public default int drink () {  
        return 20;  
    }  
  
    public default void eat () {  
        System.out.println("Cats eat");  
    }  
  
    public default int drink (); // Compile Time Error  
  
    public void eat () { // Compile Time Error  
        System.out.println("Cats eat");  
    }  
  
}
```

```
public interface Cat {  
  
    public static int drink () {  
        return 20;  
    }  
  
    public static void eat () {  
        System.out.println("Cats eat");  
    }  
  
    public static int drink (); // Compile Time Error  
  
    public void eat () { // Compile Time Error  
        System.out.println("Cats eat");  
    }  
  
}
```



8: An interface can have **public or **default** access; however, cannot be declared as **private**, **protected****

```
public interface Cat {  
}
```



```
interface Cat {  
}
```



```
private interface Cat {  
}
```



```
protected interface Cat {  
}
```



Project for Abstract Class and Interface

```
public abstract class Car {  
    public abstract void move();  
    public void Diesel(){  
        "This car works with diesel"  
    }  
    public void Gas(){  
        "This car works with gas"  
    }  
}
```

```
public interface Inside {  
    public abstract void seat();  
    public abstract void steeringWheel();  
    public abstract void radio();  
}
```

```
public interface Outside {  
    public abstract void tire();  
    public abstract void hood();  
    public abstract void headlight();  
}
```

```
public class HondaCivic extends Car implements Inside, Outside {  
    public void move(){  
        "This car can move fast in sport mode"  
    }  
    public void Gas(){  
        "This car works with gas"  
    }  
    public void seat(){  
        "This car has 5 seats"  
    }  
    public void steeringWheel(){  
        "This car has leather Steering Wheel"  
    }  
    public void engine(){  
        "This car has 1.8 VTI engine"  
    }  
    public void tire(){  
        "This car has 16 inches tires"  
    }  
    public void door(){  
        "This car has automatic door"  
    }  
    public void window(){  
        "This car has colorful windows"  
    }  
}
```



Project for Abstract Class and Interface

```
double salary(String department);  
void task();  
void personallInfo();
```

Interface Employee

```
void task(){“Automation tester”}
```

abstract class ItDepartment

```
void task(){“Sales person”}
```

abstract class MarketingDepartment

```
double salary(String department){  
    Return salary according to department and kids and education;  
}  
void personallInfo(){kids, education}
```

class Accounting



Questions

1)

Which of the following statements can be inserted in the blank line so that the code will compile successfully? (Choose all that apply)

```
public interface CanHop {}  
public class Frog implements CanHop {  
    public static void main(String[] args) {  
        _____ frog = new TurtleFrog();  
    }  
}
```

```
public class BrazilianHornedFrog extends Frog {}  
public class TurtleFrog extends Frog {}
```

- A. Frog
- B. TurtleFrog
- C. BrazilianHornedFrog
- D. CanHop
- E. Object
- F. Long



2)

Choose the correct statement about the following code:

```
1: interface HasExoskeleton {  
2:     abstract int getNumberOfSections();  
3: }  
4: abstract class Insect implements HasExoskeleton {  
5:     abstract int getNumberOfLegs();  
6: }  
7: public class Beetle extends Insect {  
8:     int getNumberOfLegs() { return 6; }  
9: }
```

- A. It compiles and runs without issue.
- B. The code will not compile because of line 2.
- C. The code will not compile because of line 4.
- D. The code will not compile because of line 7.
- E. It compiles but throws an exception at runtime.



3)

Choose the correct statement about the following code:

```
1: public interface Herbivore {  
2:     int amount = 10;  
3:     public static void eatGrass();  
4:     public int chew() {  
5:         return 13;  
6:     }  
7: }
```

- A.** It compiles and runs without issue.
- B.** The code will not compile because of line 2.
- C.** The code will not compile because of line 3.
- D.** The code will not compile because of line 4.
- E.** The code will not compile because of lines 2 and 3.
- F.** The code will not compile because of lines 3 and 4.



4)

Choose the correct statement about the following code:

```
1: public interface CanFly {  
2:     void fly();  
3: }  
4: interface HasWings {  
5:     public abstract Object getWindSpan();  
6: }  
7: abstract class Falcon implements CanFly, HasWings {  
8: }
```

- A. It compiles without issue.
- B. The code will not compile because of line 2.
- C. The code will not compile because of line 4.
- D. The code will not compile because of line 5.
- E. The code will not compile because of lines 2 and 5.
- F. The code will not compile because the class Falcon doesn't implement the interface methods.



5)

Which statements are true for both abstract classes and interfaces? (Choose all that apply)

- A.** All methods within them are assumed to be abstract.
- B.** Both can contain public static final variables.
- C.** Both can be extended using the extend keyword.
- D.** Both can contain default methods.
- E.** Both can contain static methods.
- F.** Neither can be instantiated directly.



Answers

1)

A, B, D, E. The blank can be filled with any class or interface that is a supertype of TurtleFrog. Option A is a superclass of TurtleFrog, and option B is the same class, so both are correct. BrazilianHornedFrog is not a superclass of TurtleFrog, so option C is incorrect. TurtleFrog inherits the CanHope interface, so option D is correct. All classes inherit Object, so option E is correct. Finally, Long is an unrelated class that is not a superclass of TurtleFrog, and is therefore incorrect.

2)

D. The code fails to compile because Beetle, the first concrete subclass, doesn't implement getNumberOfSections(), which is inherited as an abstract method; therefore, option D is correct. Option B is incorrect because there is nothing wrong with this interface method definition. Option C is incorrect because an abstract class is not required to implement any abstract methods, including those inherited from an interface. Option E is incorrect because the code fails at compilation-time.



3)

F. The interface variable amount is correctly declared, with public and static being assumed and automatically inserted by the compiler, so option B is incorrect. The method declaration for eatGrass() on line 3 is incorrect because the method has been marked as static but no method body has been provided. The method declaration for chew() on line 4 is also incorrect, since an interface method that provides a body must be marked as default or static explicitly. Therefore, option F is the correct answer since this code contains two compile-time errors.

4)

A. Although the definition of methods on lines 2 and 5 vary, both will be converted to public abstract by the compiler. Line 4 is fine, because an interface can have public or default access. Finally, the class Falcon doesn't need to implement the interface methods because it is marked as abstract. Therefore, the code will compile without issue.



5) B, C, E, F. Option A is wrong, because an abstract class may contain concrete methods. Since Java 8, interfaces may also contain concrete methods in form of static or default methods. Although all variables in interfaces are assumed to be public static final, abstract classes may contain them as well, so option B is correct. Both abstract classes and interfaces can be extended with the extends keyword, so option C is correct. Only interfaces can contain default methods, so option D is incorrect. Both abstract classes and interfaces can contain static methods, so option E is correct. Both structures require a concrete subclass to be instantiated, so option F is correct.



To start the test follow the steps

- 1) www.socrative.com**
- 2) Click on “Login”**
- 3) Click on “Student Login”**
- 4) Type “ALPTEKIN3523” for “Room Name”**
- 5) Click on “JOIN”**
- 6) Type your first and last name.
Note: Do not use Nick Name**

Note: You have *15 minutes* to complete the test



Iterators

1) Iterator

```
package day29_collections;

import java.util.Iterator;

public class IteratorAndForEachLoop {

    public static void main(String[] args){

        List<String> list2 = new LinkedList<String>();
        list2.add("X");
        list2.add("Y");
        list2.add("Z");
        System.out.println(list2);

        //Iterator is used to modify the collections.
        //It has three methods
        //    1)hasNext():Returns true if there are more elements.Otherwise,returns false.
        //    2)next():Returns the next element.Throws NoSuchElementException if there is not a next element.
        //    3)remove():Removes the current element.
        //          Throws IllegalStateException if you call remove( ) before using next( ).

        Iterator<String> iterator1 = list2.iterator();
        //To print elements we can use next() method
        while(iterator1.hasNext()) {
            System.out.println(iterator1.next());
        }

        //We can use for-each() loop to print elements on the console as well
        for(String w:list2) {
            System.out.println(w);
        }

        //What is the difference between for-each() and iterator() ?
        //We cannot modify a collection by using for-each(); however, iterator() can modify.
        for(String w:list2) {
            w=w+"M";
        }
        System.out.println("After for-each() loop: " + list2);

        //Before every while you need to use iterator()
        Iterator<String> iterator2 = list2.iterator();
        while(iterator2.hasNext()) {
            iterator2.next(); // Do not forget to use next() before using remove()
            iterator2.remove();
        }
        System.out.println("After iterator(): " + list2);
    }
}
```



2) ListIterator

```
package day29_collections;

import java.util.LinkedList;

public class ListIteratorMethods01 {

    public static void main(String[] args) {

        List<String> list = new LinkedList<String>();
        list.add("A");
        list.add("B");
        list.add("C");
        System.out.println("List: " + list);

        System.out.println("List is in the given order");

        //To print elements on the console
        ListIterator<String> listItr1 = list.listIterator();
        while(listItr1.hasNext()){
            Object element = listItr1.next();
            System.out.println(element);
        }

        System.out.println("List is reversed");

        //To display the list backwards use hasPrevious() and previous()
        //Before using hasPrevious() and previous(), you need to use hasNext() and next()
        //to move the pointer to the end of the list
        while(listItr1.hasPrevious()){
            Object element = listItr1.previous();
            System.out.println(element);
        }

        System.out.println("List is updated");

        //To update the elements use set()
        ListIterator<String> listItr2 = list.listIterator();
        while(listItr2.hasNext()){
            Object element = listItr2.next();
            listItr2.set(element + "W");
        }
        System.out.println("Updated list: " + list);
    }
}
```



```

package day29_collections;

import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;

public class ListIteratorMethods02 {

    public static void main(String[] args) {

        List<String> list = new LinkedList<String>();
        list.add("A");
        list.add("B");
        list.add("C");
        System.out.println("List: " + list);

        //To remove all elements use remove() after next();
        //if you do not use next you will get IllegalStateException
        ListIterator<String> listItr1 = list.listIterator();
        while(listItr1.hasNext()){
            listItr1.next();
            listItr1.remove();
        }
        System.out.println("After removing: " + list);

        //To add new elements to the list use add()
        listItr1.add("X");
        listItr1.add("Y");
        listItr1.add("Z");
        System.out.println("After adding: " + list);

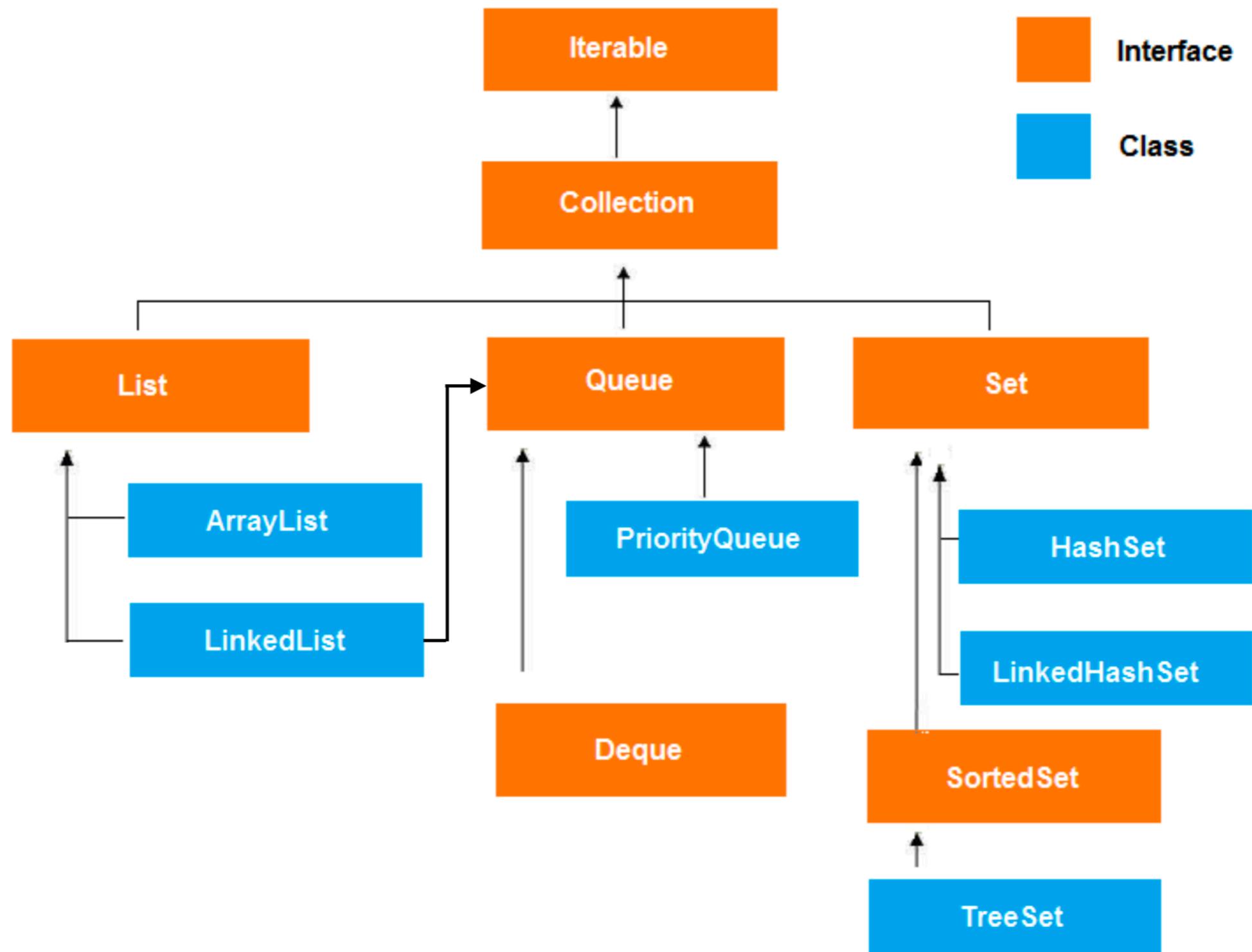
    }

}

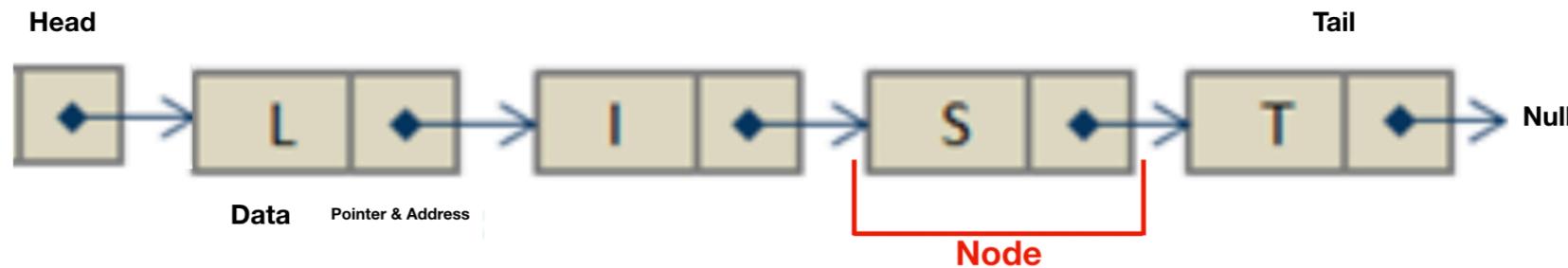
```



Collections



Linked List (Class)



- 1) Head of the LinkedList only contains the Address of the 1st element of the List.
- 2) The Last element of the LinkedList contains null in the pointer part of the node because it is the end of the List so it doesn't point to anything as shown in the above diagram.
- 3) Every element is a separate object with a **data part** and **address part**.
- 4) The elements are linked using **pointers** and **addresses**.
- 5) Each element is known as a **node**.
- 6) It is more **dynamic** than arrays and **easy to insert** and **delete** elements.
Therefore, they are preferred over the arrays.
- 7) It also has few disadvantages like the **nodes cannot be accessed directly** instead we need to start from the head and follow through the link to reach to a node we wish to access.



Note: Insert and delete operations in the Linked list are easy because adding and deleting an element from the linked list does not require element shifting, only the pointer of the previous and the next node requires change.

```
import java.util.*;
public class JavaExample{
    public static void main(String args[]){
        LinkedList<String> list=new LinkedList<String>();

        //Adding elements to the Linked list
        list.add("Steve");
        list.add("Carl");
        list.add("Raj");
        list.add("Negan");
        list.add("Rick");

        //Removing First element
        //Same as list.remove(0);
        list.removeFirst();

        //Removing Last element
        list.removeLast();

        //Iterating LinkedList
        Iterator<String> iterator=list.iterator();
        while(iterator.hasNext()){
            System.out.print(iterator.next()+" ");
        }

        //removing 2nd element, index starts with 0
        list.remove(1);

        System.out.print("\nAfter removing second element: ");
        //Iterating LinkedList again
        Iterator<String> iterator2=list.iterator();
        while(iterator2.hasNext()){
            System.out.print(iterator2.next()+" ");
        }
    }
}
```

Delete an element

```
import java.util.*;
public class JavaExample{
    public static void main(String args++){
        LinkedList<String> list=new LinkedList<String>();

        //Adding elements to the Linked list
        list.add("Steve");
        list.add("Carl");
        list.add("Raj");

        //Adding an element to the first position
        list.addFirst("Negan");

        //Adding an element to the last position
        list.addLast("Rick");

        //Adding an element to the 3rd position
        list.add(2, "Glenn");

        //Iterating LinkedList
        Iterator<String> iterator=list.iterator();
        while(iterator.hasNext()){
            System.out.println(iterator.next());
        }
    }
}
```

Insert an element



Methods of Linked List

```
LinkedList<String> llistobj = new LinkedList<String>();
```

1) **boolean add(Object item)**: It adds the item at the end of the list.

```
llistobj.add("Hello");
```

It would add the string “Hello” at the end of the linked list.

2) **void add(int index, Object item)**: It adds an item at the given index of the the list.

```
llistobj.add(2, "bye");
```

This will add the string “bye” at the 3rd position(2 index is 3rd position as index starts with 0).

3) **boolean addAll(Collection c)**: It adds all the elements of the specified collection c to the list. It throws NullPointerException if the specified collection is null. Consider the below example –

```
LinkedList<String> llistobj = new LinkedList<String>();
ArrayList<String> arraylist= new ArrayList<String>();
arraylist.add("String1");
arraylist.add("String2");
llistobj.addAll(arraylist);
```

This piece of code would add all the elements of ArrayList to the LinkedList.

4) **boolean addAll(int index, Collection c)**: It adds all the elements of collection c to the list starting from a give index in the list. It throws NullPointerException if the collection c is null and IndexOutOfBoundsException when the specified index is out of the range.

```
llistobj.add(5, arraylist);
```

It would add all the elements of the ArrayList to the LinkedList starting from position 6 (index 5).



5) void addFirst(Object item): It adds the item (or element) at the first position in the list.

```
llistobj.addFirst("text");
```

It would add the string “text” at the beginning of the list.

6) void addLast(Object item): It inserts the specified item at the end of the list.

```
llistobj.addLast("Chaitanya");
```

This statement will add a string “Chaitanya” at the end position of the linked list.

7) void clear(): It removes all the elements of a list.

```
llistobj.clear();
```

8) Object clone(): It returns the copy of the list.

For e.g. My linkedList has four items: text1, text2, text3 and text4.

```
Object str= llistobj.clone();
System.out.println(str);
```

Output: The output of above code would be:

[text1, text2, text3, text4]

9) boolean contains(Object item): It checks whether the given item is present in the list or not. If the item is present then it returns true else false.

```
boolean var = llistobj.contains("TestString");
```

It will check whether the string “TestString” exist in the list or not.



10) **Object get(int index):** It returns the item of the specified index from the list.

```
Object var = llistobj.get(2);
```

It will fetch the 3rd item from the list.

11) **Object getFirst():** It fetches the first item from the list.

```
Object var = llistobj.getFirst();
```

12) **Object getLast():** It fetches the last item from the list.

```
Object var= llistobj.getLast();
```

13) **int indexOf(Object item):** It returns the index of the specified item.

```
llistobj.indexOf("bye");
```

14) **Object remove():** It removes the first element of the list.

```
llistobj.remove();
```

15) **Object remove(int index):** It removes the item from the list which is present at the specified index.

```
llistobj.remove(4);
```

It will remove the 5th element from the list.

16) **Object remove(Object obj):** It removes the specified object from the list.

```
llistobj.remove("Test Item");
```

17) **Object removeFirst():** It removes the first item from the list.

```
llistobj.removeFirst();
```

18) **Object removeLast():** It removes the last item of the list.

```
llistobj.removeLast();
```



19) Object set(int index, Object item): It updates the item of specified index with the give value.

```
llistobj.set(2, "Test");
```

It will update the 3rd element with the string “Test”.

20) int size(): It returns the number of elements of the list.

```
llistobj.size();
```

21) Collections.sort(list) : It sorts the list elements in natural order



Set (Interface)

A Set is a Collection that cannot contain duplicate elements.

Note: There are three main implementations of Set interface:

- 1) HashSet
- 2) TreeSet
- 3) LinkedHashSet.

1) HashSet :

HashSet stores its elements in a hash table.

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

Note: Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.

Like in universities, each student is assigned a unique roll number that can be used to retrieve information about them.

{“abcdef”, “bcdefa”, “cdefab” , “defabc” }

HashTable : A hash table is a data structure that is used to store key/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched.



Note: HashSet **doesn't maintain any order**, the elements would be returned in any random order.

Note: HashSet **doesn't allow duplicates**. If you try to add a duplicate element in HashSet, the old value would be overwritten.

Note: HashSet **allows null values**; however, if you insert more than one nulls it would still return only one null value.

```
import java.util.HashSet;
public class HashSetExample {
    public static void main(String args[]) {
        // HashSet declaration
        HashSet<String> hset =
            new HashSet<String>();

        // Adding elements to the HashSet
        hset.add("Apple");
        hset.add("Mango");
        hset.add("Grapes");
        hset.add("Orange");
        hset.add("Fig");
        //Addition of duplicate elements
        hset.add("Apple");
        hset.add("Mango");
        //Addition of null values
        hset.add(null);
        hset.add(null);

        //Displaying HashSet elements
        System.out.println(hset);
    }
}
```



```

import java.util.TreeSet;
public class TreeSetExample {
    public static void main(String args[]) {
        // TreeSet of String Type
        TreeSet<String> tset = new TreeSet<String>();

        // Adding elements to TreeSet<String>
        tset.add("ABC");
        tset.add("String");
        tset.add("Test");
        tset.add("Pen");
        tset.add("Ink");
        tset.add("Jack");

        //Displaying TreeSet
        System.out.println(tset);

        // TreeSet of Integer Type
        TreeSet<Integer> tset2 = new TreeSet<Integer>();

        // Adding elements to TreeSet<Integer>
        tset2.add(88);
        tset2.add(7);
        tset2.add(101);
        tset2.add(0);
        tset2.add(3);
        tset2.add(222);
        System.out.println(tset2);
    }
}

```

2) TreeSet :

TreeSet is similar to HashSet except that it sorts the elements in the ascending order while HashSet doesn't maintain any order.

Note: HashSet is faster than TreeSet for the operations like add(), remove(), contains(), size() etc.

Note: TreeSet does not allow duplication

Note: If you want a sorted Set then it is better to add elements to HashSet and then convert It to TreeSet rather than creating a TreeSet and adding elements to it.



How to Convert a HashSet to TreeSet

```
import java.util.HashSet;
import java.util.TreeSet;
import java.util.Set;
class ConvertHashSetToTreeSet{
    public static void main(String[] args) {
        // Create a HashSet
        HashSet<String> hset = new HashSet<String>();
        ...
        //add elements to HashSet
        hset.add("Element1");
        hset.add("Element2");
        hset.add("Element3");
        hset.add("Element4");

        // Displaying HashSet elements
        System.out.println("HashSet contains: " + hset);

        // Creating a TreeSet of HashSet elements
        Set<String> tset = new TreeSet<String>(hset);

        // Displaying TreeSet elements
        System.out.println("TreeSet contains: ");
        for(String temp : tset){
            System.out.println(temp);
        }
    }
}
```



3) **LinkedHashSet :**

LinkedHashSet maintains the **insertion order**. Elements get sorted in the same sequence in which they have been added to the Set.

```
import java.util.LinkedHashSet;
public class LinkedHashSetExample {
    public static void main(String args[]) {
        // LinkedHashSet of String Type
        LinkedHashSet<String> lhset = new LinkedHashSet<String>();

        // Adding elements to the LinkedHashSet
        lhset.add("Z");
        lhset.add("PQ");
        lhset.add("N");
        lhset.add("O");
        lhset.add("KK");
        lhset.add("FGH");
        System.out.println(lhset);

        // LinkedHashSet of Integer Type
        LinkedHashSet<Integer> lhset2 = new LinkedHashSet<Integer>();

        // Adding elements
        lhset2.add(99);
        lhset2.add(7);
        lhset2.add(0);
        lhset2.add(67);
        lhset2.add(89);
        lhset2.add(66);
        System.out.println(lhset2);
    }
}
```

Output

[Z, PQ, N, O, KK, FGH]

Output

[99, 7, 0, 67, 89, 66]



Queue (Interface)

```
import java.util.*;
public class QueueExample1 {

    public static void main(String[] args) {

        /*
         * We cannot create instance of a Queue as it is an
         * interface, we can create instance of LinkedList or
         * PriorityQueue and assign it to Queue
         */
        Queue<String> q = new LinkedList<String>();

        //Adding elements to the Queue
        q.add("Rick");
        q.add("Maggie");
        q.add("Glenn");
        q.add("Negan");
        q.add("Daryl");

        System.out.println("Elements in Queue:"+q);
    }

    /*
     * We can remove element from Queue using remove() method,
     * this would remove the first element from the Queue
     */
    System.out.println("Removed element: "+q.remove());
}

/*
 * element() method - this returns the head of the
 * Queue. Head is the first element of Queue
 */
System.out.println("Head: "+q.element());

/*
 * poll() method - this removes and returns the
 * head of the Queue. Returns null if the Queue is empty
 */
System.out.println("poll(): "+q.poll());

/*
 * peek() method - it works same as element() method,
 * however it returns null if the Queue is empty
 */
System.out.println("peek(): "+q.peek());

//Again displaying the elements of Queue
System.out.println("Elements in Queue:"+q);
}
```

Elements added to Queue are placed at the end and removed from the beginning of Queue.

FIFO (First In First Out)

Queue Examples:

- 1) Ticket counter line where people who come first will get his ticket first.
 - 2) Bank line where people who come first will done his transaction first.
 - 3) In a multitasking operating system, the CPU cannot run all jobs at once, so jobs must be batched up and then scheduled according to some policy.
- A queue might be a suitable option in this case.

Elements in Queue:[Rick, Maggie, Glenn, Negan, Daryl]

Removed element: Rick

Head: Maggie

poll(): Maggie

peek(): Glenn

Elements in Queue:[Glenn, Negan, Daryl]



Difference Between Queue created by using LinkedList Class and Queue created by using PriorityQueue Class

```
public class QueueInJava {  
  
    public static void main(String[] args) {  
        Queue<String> queue = new LinkedList<String>();  
        queue.add("Ishfaq");  
        queue.add("Ramzan");  
        queue.add("Nagoo");  
        queue.add("Bangalore");  
  
    }
```

```
System.out.println("Linked List Queue is:"+ queue);  
System.out.println("Linked List Queue Peek is :" +queue.peek());  
  
queue.poll();  
System.out.println("Linked List Queue after remove is:"+ queue);
```

```
Queue<Integer> queuenew = new PriorityQueue<Integer>();  
  
queuenew.add(2);  
queuenew.add(3);  
queuenew.add(1);  
queuenew.add(0);  
queuenew.add(4);  
  
queuenew.add(5);  
  
System.out.println("Priority Queue is:"+ queuenew);  
System.out.println("Priority Queue Peek is :" +queuenew.peek());  
  
int ieleFirst=queuenew.remove();  
System.out.println("Priority Queue Element Removed is:"+ ieleFirst);  
int ieleSecond=queuenew.remove();  
System.out.println("Priority Queue Element Removed is:"+ ieleSecond);  
System.out.println("Priority Queue after remove is:"+ queuenew);  
}  
}
```

If you create Queue by **LinkedList** Class, elements get sorted in the same sequence in which they have been added to the Set.

If you create Queue by **PriorityQueue** Class, it orders the element in ascending order.

Linked List Queue is:[Ishfaq, Ramzan, Nagoo, Bangalore]

Linked List Queue Peek is :Ishfaq

Linked List Queue after remove is:[Ramzan, Nagoo, Bangalore]

Priority Queue is:[0, 1, 2, 3, 4]

Priority Queue Peek is :0

Priority Queue Element Removed is:0

Priority Queue Element Removed is:1

Priority Queue after remove is:[2, 3, 4]



Deque (Interface)

- 1) The Deque interface is a subtype of the Queue interface.
- 2) The Deque is a Double-ended queue that supports adding and removing elements from both ends (*head and tail*) of the data structure. It can be used in FIFO (First In First Out) or in LIFO (Last In First Out)
- 3) Deque is faster than LinkedList.
- 4) Deques are resizable.
- 5) Deques do not accept Null as an element.



```

public class DequeExample
{
    public static void main(String[] args)
    {
        Deque<String> deque = new LinkedList<String>();

        // We can add elements to the queue in various ways
        deque.add("Element 1 (Tail)"); // add to tail
        deque.addFirst("Element 2 (Head)");
        deque.addLast("Element 3 (Tail)");
        deque.push("Element 4 (Head)"); //add to head
        deque.offer("Element 5 (Tail)");
        deque.offerFirst("Element 6 (Head)");
        deque.offerLast("Element 7 (Tail)");

        System.out.println(deque + "\n");

        // Iterate through the queue elements.
        System.out.println("Standard Iterator");
        Iterator iterator = deque.iterator();
        while (iterator.hasNext())
            System.out.println("\t" + iterator.next());

        // Reverse order iterator
        Iterator reverse = deque.descendingIterator();
        System.out.println("Reverse Iterator");
        while (reverse.hasNext())
            System.out.println("\t" + reverse.next());

        // Peek returns the head, without deleting
        // it from the deque
        System.out.println("Peek " + deque.peek());
        System.out.println("After peek: " + deque);

        // Pop returns the head, and removes it from
        // the deque
        System.out.println("Pop " + deque.pop());
        System.out.println("After pop: " + deque);

        // We can check if a specific element exists
        // in the deque
        System.out.println("Contains element 3: " +
                           deque.contains("Element 3 (Tail)"));

        // We can remove the first / last element.
        deque.removeFirst();
        deque.removeLast();
        System.out.println("Deque after removing " +
                           "first and last: " + deque);
    }
}

```

The difference is that `offer()` will return **false** if it fails to insert the element on a size restricted Queue, whereas `add()` will throw an **IllegalStateException**.

You should use `offer()` when failure to insert an element would be normal, and `add()` when failure would be an exceptional occurrence



```
Deque<Integer> arrayDeque = new ArrayDeque<>();  
  
arrayDeque.add(3);  
arrayDeque.push(4);  
arrayDeque.offer(6);  
arrayDeque.addFirst(2);  
arrayDeque.addLast(5);  
arrayDeque.addFirst(1);  
System.out.println("ArrayDeque: " + arrayDeque.toString());
```

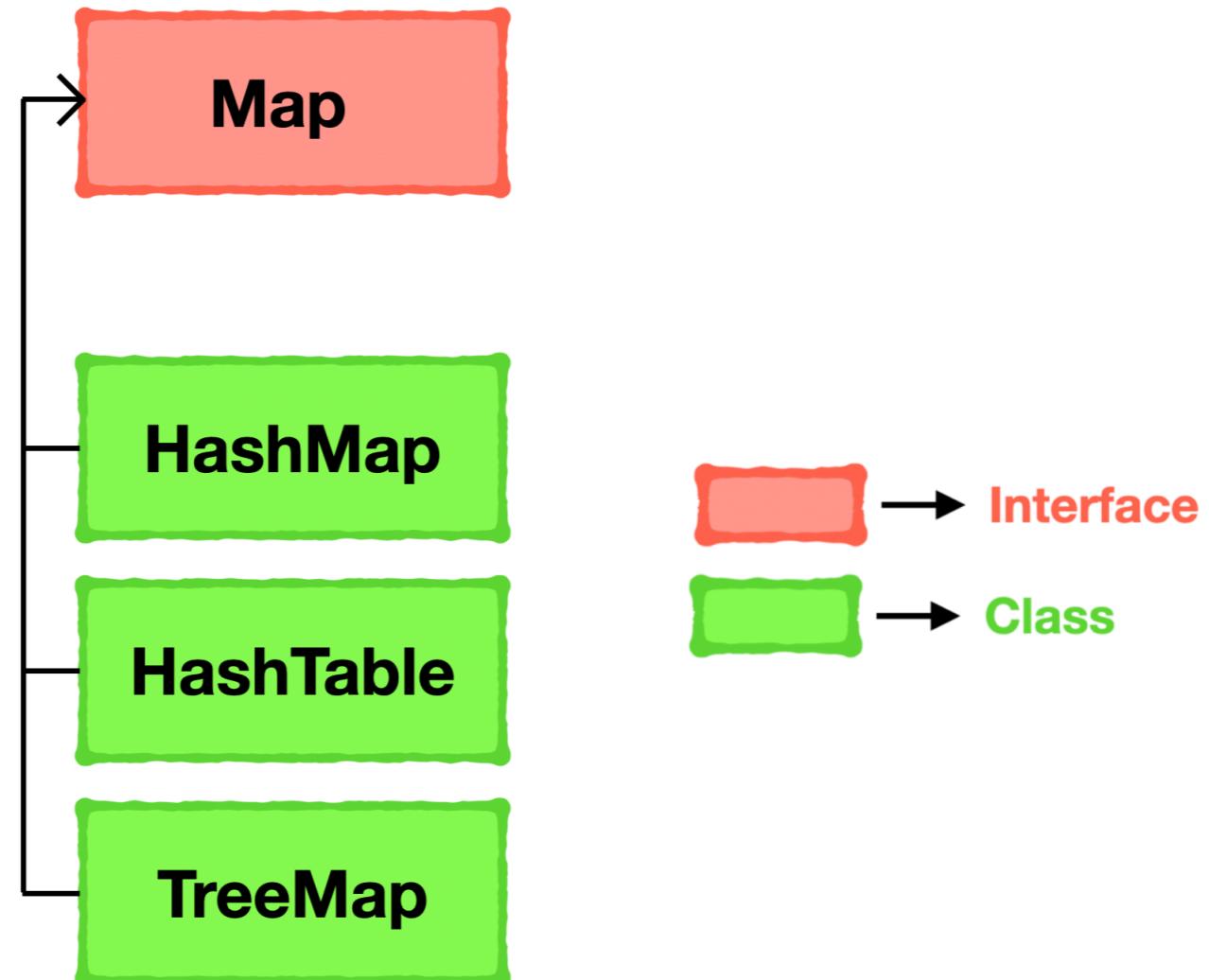
The output is:

```
ArrayDeque: [1, 2, 4, 3, 6, 5]
```

```
// Add 3 at the tail of this deque  
arrayDeque.add(3); -> [3]  
// Add 4 at the head of this deque  
arrayDeque.push(4); -> [4, 3]  
// Add 6 at the tail of this deque  
arrayDeque.offer(6); -> [4, 3, 6]  
// Add 2 at the head of this deque  
arrayDeque.addFirst(2); -> [2, 4, 3, 6]  
// Add 5 at the tail of this deque  
arrayDeque.addLast(5); -> [2, 4, 3, 6, 5]  
// Add 1 at the head of this deque  
arrayDeque.addFirst(1); -> [1, 2, 4, 3, 6, 5]
```



Maps



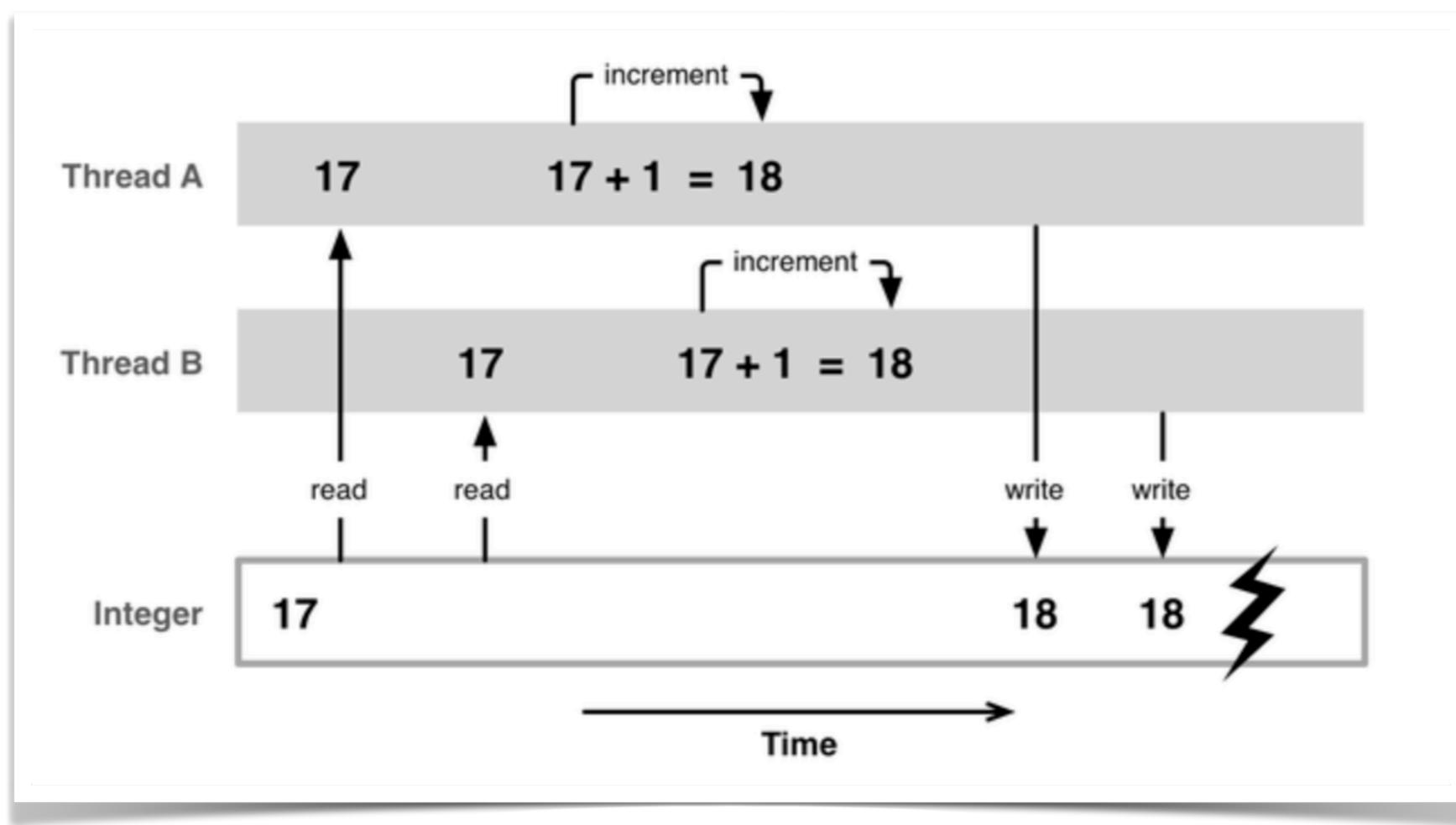
Multithreading

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

Each part of such program is called a thread.

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface



Synchronizing

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.



Table

In case of duplicate key

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	

In this case key will be old one only (old mohan) value will be changed(156 will replaced with 159).

hashcode	Key	Value	next node
104075143	mohan	159	null

hashcode	Key	Value	next node
45720776	0-42L	156	null

In case of hash collision

hashcode	Key	value
112671	ram	158



HashMap

- 1) HashMap is **not-synchronized**. It is **not-thread safe** and can't be shared between many threads
- 2) HashMap allows **one null key** and **multiple null values**.
- 3) HashMap is **fast**
- 4) HashMap is **not ordered**

```
public static void main(String[] args) {  
  
    HashMap<Integer, String> hashmap = new HashMap<Integer, String>();  
    hashmap.put(100, "Mark"); //adding key value pairs  
    hashmap.put(101, "Angie");  
    hashmap.put(102, "Banu");  
    // You can enter multiple null key but it accepts just the last one  
    hashmap.put(null, "Brad");  
    hashmap.put(null, "Ali");  
    // You can enter multiple null value and it accepts all  
    hashmap.put(3, null);  
    hashmap.put(1, null);  
    System.out.println(hashmap); // Prints the list of key,value pairs  
}
```

{null=Ali, 1=null, 3=null, 100=Mark, 101=Angie, 102=Banu}

HashTable

- 1) HashTable is **synchronized**. It is **thread safe** and can be shared between many threads
- 2) HashTable **doesn't allow any null key or value**.
- 3) HashTable is **slow**
- 4) HashTable is **not ordered**

```
public static void main(String[] args) {  
  
    Hashtable<Integer, String> hashTable = new Hashtable<>();  
    //Adding key value pairs  
    hashTable.put(101, "Mark");  
    hashTable.put(100, "Angie");  
    hashTable.put(102, "John");  
    //You cannot enter any null key  
    //You cannot enter any null value  
    System.out.println(hashTable);  
}
```

{102=John, 101=Mark, 100=Angie}

TreeMap

- 1) TreeMap is **not-synchronized**. It is **not-thread safe** and can't be shared between many threads
- 2) TreeMap **does not allow null key** but allows **multiple null values**.
- 3) TreeMap is **slow**
- 4) TreeMap keeps its **entries sorted** according to the **natural ordering of its keys**.
For numbers ascending order, for strings alphabetical order.

```
public static void main(String[] args) {  
  
    TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();  
    // Adding key value pairs  
    treeMap.put(27, "Geeks");  
    treeMap.put(15, "4");  
    treeMap.put(50, "Geeks");  
    treeMap.put(25, "Welcomes");  
    treeMap.put(30, "You");  
    //Does not allow null keys and thus a NullPointerException is thrown.  
    //However, multiple null values can be associated with different keys.  
    treeMap.put(40, null);  
    treeMap.put(33, null);  
  
    // Printing the TreeMap  
    System.out.println("TreeMap: " + treeMap);  
}
```

{15=4, 25=Welcomes, 27=Geeks, 30=You, 33=null, 40=null, 50=Geeks}



More About HashMap

```
public class HashMap01 {  
    public static void main(String[] args) {  
  
        HashMap<Integer, String> hashMap = new HashMap<Integer, String>();  
        //put(Key k, Value v) inserts key value mapping into the map  
        hashMap.put(100, "Mark"); // {100 = Mark}  
        hashMap.put(101, "Angie"); // {100 = Mark, 101 = Angie}  
        hashMap.put(102, "Banu"); // {100 = Mark, 101 = Angie, 102 = Banu}  
        System.out.println(hashMap); // {100 = Mark, 101 = Angie, 102 = Banu}  
  
        // remove(Object key) removes the key-value pair for the specified key  
        hashMap.remove(101);  
        System.out.println("After using remove(): " + hashMap);  
  
        //get(Object key) returns the value for the specified key.  
        System.out.println(hashMap.get(102)); // Banu  
  
        //containsKey(Object key): It is a boolean function which returns true or false  
        //based on whether the specified key is found in the map.  
        System.out.println(hashMap.containsKey(100)); // true  
        System.out.println(hashMap.containsKey(101)); // false  
  
        //isEmpty() is a boolean function which returns true or false  
        //It checks whether the map is empty.  
        System.out.println(hashMap.isEmpty()); // false  
  
        //keySet() returns the Set of the keys fetched from the map.  
        System.out.println(hashMap.keySet()); // [100, 102]  
  
        //values() returns a collection of values of map.  
        System.out.println(hashMap.values()); // [Mark, Banu]  
  
        //size() Returns the size of the map  
        System.out.println(hashMap.size()); // 2  
  
        // clear() removes all the key and value pairs from the specified Map.  
        hashMap.clear();  
        System.out.println(hashMap); // {}  
    }  
}
```



More About HashMap

```
public class HashMap02 {  
    public static void main(String[] args) {  
  
        HashMap<Integer, String> hashMap = new HashMap<Integer, String>();  
        //Adding elements to HashMap  
        hashMap.put(11, "AB");  
        hashMap.put(2, "CD");  
        hashMap.put(33, "EF");  
        hashMap.put(9, "GH");  
        hashMap.put(3, "IJ");  
  
        //FOR LOOP  
        System.out.println("For Loop:");  
        for (HashMap.Entry me1 : hashMap.entrySet()) {  
            System.out.println("Key: "+me1.getKey() + " & Value: " + me1.getValue());  
        }  
  
        //WHILE LOOP & ITERATOR  
        System.out.println("While Loop:");  
        Iterator iterator = hashMap.entrySet().iterator();  
        while (iterator.hasNext()) {  
            HashMap.Entry me2 = (HashMap.Entry) iterator.next();  
            System.out.println("Key: "+me2.getKey() + " & Value: " + me2.getValue());  
        }  
    }  
}
```



More About HashTable

```
public class HashTable01 {  
    public static void main(String[] args) {  
  
        Hashtable<Integer, String> hashTable = new Hashtable<>();  
  
        hashTable.put(100, "Mark"); // {100=Mark}  
        hashTable.put(101, "Amanda"); // {101=Amanda, 100=Mark}  
        hashTable.put(102, "Banu"); // {102=Banu, 101=Amanda, 100=Mark}  
        System.out.println(hashTable); // {102=Banu, 101=Amanda, 100=Mark}  
  
        // remove(Object key) removes the key-value pair for the specified key  
        hashTable.remove(101);  
        System.out.println("After using remove(): " + hashTable);  
  
        //get(Object key) returns the value for the specified key.  
        System.out.println(hashTable.get(102)); // Banu  
  
        //containsKey(Object key): It is a boolean function which returns true or false  
        //based on whether the specified key is found in the map.  
        System.out.println(hashTable.containsKey(100)); // true  
        System.out.println(hashTable.containsKey(101)); // false  
  
        //isEmpty() is a boolean function which returns true or false  
        //It checks whether the map is empty.  
        System.out.println(hashTable.isEmpty()); // false  
  
        //keySet() returns the Set of the keys fetched from the map.  
        System.out.println(hashTable.keySet()); // [100, 102]  
  
        //values() returns a collection of values of map.  
        System.out.println(hashTable.values()); // [Banu, Mark]  
  
        //size() Returns the size of the map  
        System.out.println(hashTable.size()); // 2  
  
        // clear() removes all the key and value pairs from the specified Map.  
        hashTable.clear();  
        System.out.println(hashTable); // {}  
    }  
}
```



More About HashTable

```
import java.util.Enumeration;[]

public class HashTable02 {

    public static void main(String[] args) {

        // Creating a HashTable
        Hashtable<String, String> hashTable = new Hashtable<String, String>();

        // Adding Key and Value pairs to HashTable
        hashTable.put("Key1", "Chaitanya");
        hashTable.put("Key2", "Ajeet");
        hashTable.put("Key3", "Peter");
        hashTable.put("Key4", "Ricky");

        // What is Enumeration?
        // Enumeration is an interface and found in the java.util package.
        // The main difference between Iterator and Enumeration is removal of the element while traversing the collection.
        // Iterator can remove the element during traversal of collection as it has remove() method.
        // Enumeration does not have remove() method.
        // As you know; Iterator has hasNext(), next(), remove() methods.
        // Enumeration has hasMoreElements() and nextElement() methods, Enumeration does not have remove() method.

        Enumeration names = hashTable.keys();
        while (names.hasMoreElements()) {
            String key = (String) names.nextElement();
            System.out.println("Key: " + key + " & Value: " + hashTable.get(key));
        }
    }
}
```



More About TreeMaps

```
public class TreeMap01 {  
  
    public static void main(String[] args) {  
  
        //This is how to declare TreeMap  
        TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();  
  
        //put() adds elements to TreeMap  
        treeMap.put(1, "Mark");  
        treeMap.put(43, "Amanda");  
        treeMap.put(35, "John");  
        System.out.println(treeMap);  
  
        // remove(Object key) removes the key-value pair for the specified key  
        treeMap.remove(101);  
        System.out.println("After using remove(): " + treeMap);  
  
        //get(Object key) returns the value for the specified key.  
        System.out.println(treeMap.get(102)); // Banu  
  
        //containsKey(Object key): It is a boolean function which returns true or false  
        //based on whether the specified key is found in the map.  
        System.out.println(treeMap.containsKey(100)); // true  
        System.out.println(treeMap.containsKey(101)); // false  
  
        //isEmpty() is a boolean function which returns true or false  
        //It checks whether the map is empty.  
        System.out.println(treeMap.isEmpty()); // false  
  
        //keySet() returns the Set of the keys fetched from the map.  
        System.out.println(treeMap.keySet()); // [100, 102]  
  
        //values() returns a collection of values of map.  
        System.out.println(treeMap.values()); // [Mark, Banu]  
  
        //size() Returns the size of the map  
        System.out.println(treeMap.size()); // 2  
  
        // clear() removes all the key and value pairs from the specified Map.  
        treeMap.clear();  
        System.out.println(treeMap); // {}  
    }  
}
```



More About TreeMaps

```
public class TreeMap02 {  
  
    public static void main(String[] args) {  
  
        //This is how to declare TreeMap  
        TreeMap<Integer, String> treeMap = new TreeMap<Integer, String>();  
  
        //put() adds elements to TreeMap  
        treeMap.put(1, "Mark"); //1=Mark  
        treeMap.put(43, "Amanda"); //1=Mark, 43=Amanda  
        treeMap.put(35, "John"); //1=Mark, 35=John, 43=Amanda  
        System.out.println(treeMap); //1=Mark, 35=John, 43=Amanda}  
  
        // Display content using Iterator  
        Iterator iterator = treeMap.entrySet().iterator();  
        while(iterator.hasNext()) {  
            // Map.Entry interface in Java provides certain methods to access the entry in the Map.  
            // By gaining access to the entry of the Map we can easily manipulate them.  
            // Map.Entry is a generic and is defined in the java.util package.  
            Map.Entry mapEntry = (Map.Entry)iterator.next();  
            System.out.println("Key:" + mapEntry.getKey() + " Value:" + mapEntry.getValue());  
        }  
    }  
}
```



Project Algorithm (Pseudo Code)

- 1) Greet the customer**
- 2) Tell the customer about instructions**
- 3) Set account balance to \$10**
- 4) Display the account balance**
- 5) Display the products and their prices and numbers (1-10)**
- 6) Get user's selection number**
 - If the number is valid go to number 8
 - If the number is invalid go to number 7
- 7) Tell customer that their choice was invalid and go to number 4**
- 8) Set the product's price**
- 9) See if the customer has enough money for the products**
 - If yes go to number 10
 - If no go to number 13
- 10) Subtract the price from the balance**
- 11) Display the remaining balance**
- 12) Ask the customer if the customer would like to keep shopping**
 - If yes go to number 4
 - If no go to number 17
- 13) Ask the user if they would like to add money into the account**
 - If yes go to number 14
 - If no go to number 17
- 14) Ask the amount to add in the account**
- 15) Get the amount and add it to the current balance**
- 16) Display the current balance and go to number 5**
- 17) Display the current balance**
- 18) Tell the customer to have a good day**



Unified Modeling Language Diagram

UML Diagrams are used to visualize the system

Benefits of UML

- 1: Simplifies complex software design.**
- 2: It reduces thousands of words of explanation in a few graphical diagrams that may reduce time consumption to understand.**
- 3: It makes communication more clear and real.**
- 4: It helps to acquire the entire system in a view.**
- 5: It becomes easy for the software programmer to implement the actual demand because they have the clear picture of the problem.**



UML Diagram

