Repetition and Q/A

The Exam I

- Circa 50% multiple choice questions and 50% coding
- Question do not give minus points
  - Exception – Multiple answer questions (give 0.5 minus for the specific question not for the overall score!)
  - Similar in style to what we did with the Kahoot
- Coding part is split in sub parts and each part gives points
  - Pseudo code is fine if you do not remember exact syntax but try to be precise!
  - If you are not sure write your assumptions as a comment!
  - Similar to what was given in Assignment 5-10 in Canvas (without the math ;))

The Exam II

- No mathematics
- Not harder than assignments from Canvas (focus on assignment 5 upwards)
- Pseudo code is fine as long as I can see that you understood the basic concepts (e.g., if there is an inheritance example and you know that you have to use extends, etc.)
  - Important: The more precise the easier to judge if it was correct!
- I will not deduct points for small syntax errors (wrong () or {} for example). But, if you forget extends in an inheritance question for example then its wrong.

The Exam III

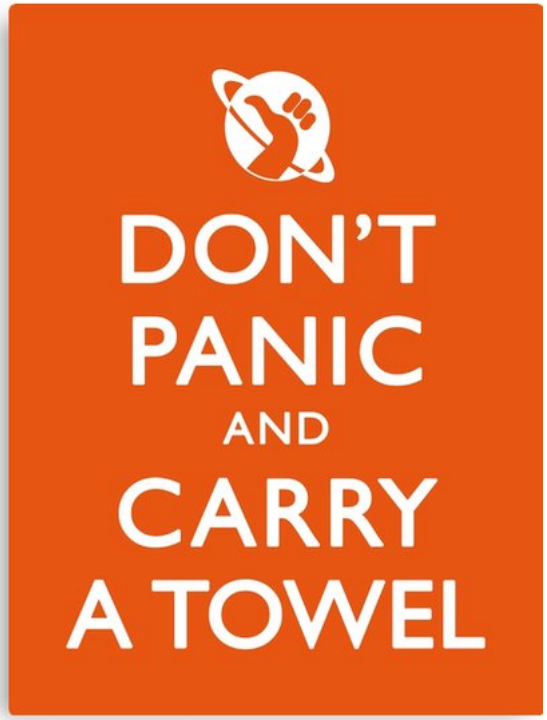
- Basics
  - Some easy questions, opportunity to get points
- Classes
  - What is a class, how to declare it, how to use it
- Objects
  - What is an object, how to declare it, how to use it
- Constructors
  - What is it, how to declare it, how to use it, constructor chaining
- Static key word
  - What is it, how to use it
- This key word
  - What is it, how to use it
- Super key word
  - What is it, how to use it

The Exam IV

- Inheritance
  - What is inheritance, what is it used for, implement simple examples (what the did for the assignments Shapes, Employees, Animals)
- Aggregation
  - What is it, what is it used for, implement simple examples (Person has address, Telephonebook has entires like in the assignments)
- Method overloading and overriding
  - What is it, how to do it, simple examples
- Polymorphism
  - What is it, how to use it, simple examples
- Abstract classes
  - What is it, how to use it, simple examples
- Interfaces
  - What is it, how to use it, simple examples
- Encapsulation

The Exam V

- Example multiple choice questions on canvas
- Example coding questions on canvas
- Don't panic ☺



What you should study!!!


Basics
Datatypes, conditions, loops and methods

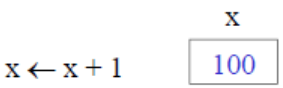
Recommended readings

- Datatypes: https://www.geeksforgeeks.org/data-types-in-java/
- Variables: https://www.geeksforgeeks.org/variables-in-java/
- Methods: https://www.geeksforgeeks.org/methods-in-java/
- Method overloading: https://www.geeksforgeeks.org/overloading-in-java/
- Loops: https://www.geeksforgeeks.org/loops-in-java/
- Conditions: https://www.geeksforgeeks.org/decision-making-javaif-else-switch-break-continue-jump/

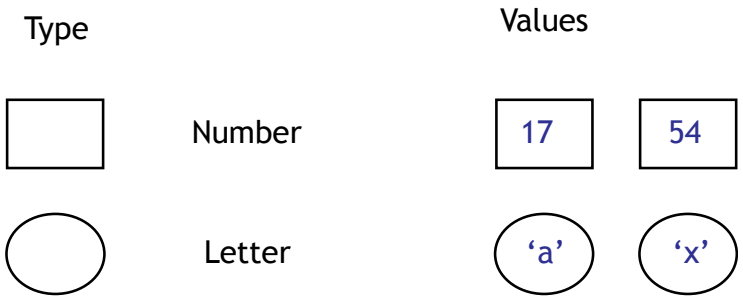
Variables

- Variables are named container for values.

- Values can change



- Variables have a data type
  - which is the set/range of values allowed for a variable.



Declaration of variables

- Each variable must be declared before use
  - Name and type are given to the compiler
  - Compiler allocates memory
- Examples:
  - `int x;` … declares variable x of type int (integer)
  - `short a, b;` … declares two variables of type short (short integer)

Integer types

byte	8 bit	-2^7 .. 2^7-1	(-128 .. 127)
short	16 bit	-2^15 .. 2^15-1	(-32.768 .. 32.767)
int	32 bit	-2^31 .. 2^31-1	(-2.147.483.648 ..)
long	64 bit	…	…

- Declaration & initialisation
  - `int x = 100;`
    declares integer x and assign value of 100.
  - `short a = 0, b = 1;`
    declares two short variables with initial values.

Constants

- Init variables that cannot be changed later
  - static final int max = 100;
- Why would you do that?
  - readability
    - max easier to read than 100
  - maintainability
    - if the same value is used several times.
- Constants are declared in class scope
  - will be explained later in the course

Arithmetic Expressions

- Simplified grammar

```
Expr = Operand {BinaryOperator Operand}.
Operand = [UnaryOperator] ( identifier | number | "(" Expr ")" ).
```

- eg. `- x + 3 * (y + 1)`

Arithmetic Expressions

- Binary Operators

+	sum	
-	subtraction	
*	multiplikation	
/	division	`5/3 = 1`
%	modulo	`5%3 = 2`

- Unary operators

+	identity	`(+x) = x`
-	invert sign	

Types in Arithmetic Expressions

- Order of operations
  - multiplication and division (*, /, %) over addition and subtraction (+, -)
    - eg. 2 + 3 * 4 = 14
  - left association
    - eg. 7 - 3 - 2 = 2
  - unary operators over binaray operators
    - eg.: -2 * 4 + 3 ergibt -5
- Resulting types
  - input type can be byte, short, int, long
  - resulting type
    - if one operand is long -> result is type long,
    - otherwise -> type int

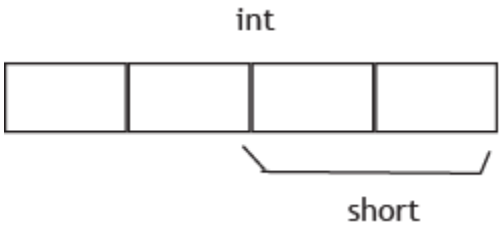
Examples

```
short s; int i; long x;
x = x + i;              // long
i = s + 1;              // int (1 is int)
s = s + 1;              // false!
s = (short)(s + 1);     // type cast necessary
```

Type Cast

`(type) expression`

- changes *expression* to *type*
- result can be truncated



Increment / Decrement

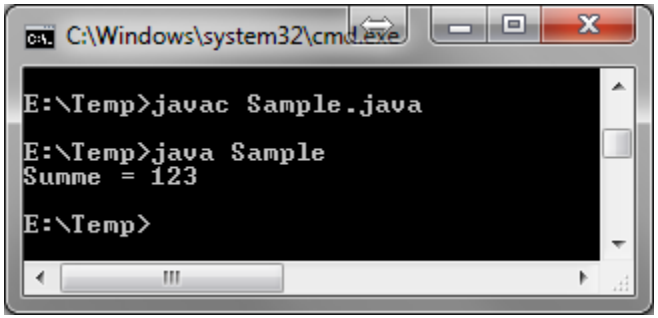
- access variable plus operation
  - x++ ... returns x and then adds +1
  - ++x ... adds 1 to x and then returns x
  - x-- , --x ... the same with subtraction.
- can be a statement on ist own right
  - x = 1; x++; // x = 2 the same as: x = x + 1;
- examples
  - x = 1; y = x++ * 3; // x = 2, y = 3 is: y = x * 3; x = x + 1;
  - x = 1; y = ++x * 3; // x = 2, y = 6 is: x = x + 1; y = x * 3;
- only works on variables, not expressions.
  - y = (x + 1)++; // wrong!

Java-Programs

```
class ProgramName {
    public static void main (String[] arg) {
            ... // Declarations
            ... // Statements
    }
}
```

Text has to be in file named *ProgramName.java*

```
// Example:
class Sample {
    public static void main (String[] arg) {
            int a = 23;
            int b = 100;
            System.out.print("Sum = ");
            System.out.println(a + b);
    }
}
```



- public : it is a access specifier that means it will be accessed by publicly.
- static : it is access modifier that means when the java program is load then it will create the space in memory automatically.
- void : it is a return type i.e it does not return any value.
- main() : it is a method or a function name.
- string args[] : its a command line argument it is a collection of variables in the string format.

Review: Data Types

- Integer Types: `byte, short, int, long`
- Floating point types: `float, double`
- Characters / Text: char, `String`
- Boolean: `boolean`

See also https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Review: Data Types

- Integer expressions are of type `int`
  - ie. byte, short, ...
- Floating point and scientific number expressions are type `double`
- Explicit type with suffix
  - „L“ or „l“ -> `long`
  - „d“ -> `double`
  - „f“ -> `float`

Review: Data Types

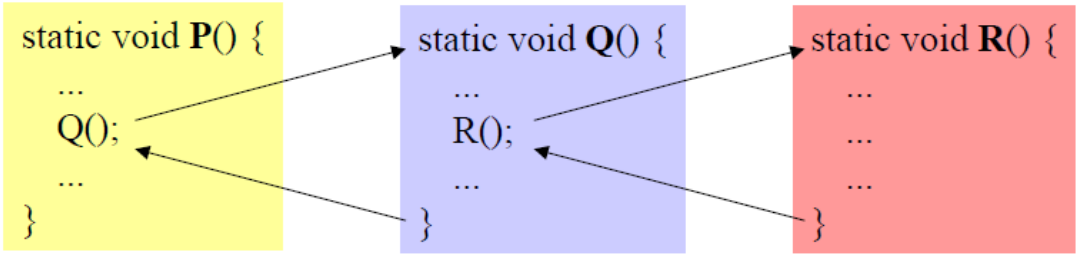
Data Type	Default Value (for fields)
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
String (or any object)	null
boolean	false

Methods

- Core of functional programming languages
  - subroutines, functions, ...
- Goal is to re-use code
  - Code that would otherwise show up more than once.
- All in all: less to write
  - less lines of code, less work
  - easier to find errors and maintain.

Methods in Java

- Can be functions or procedures (sub routines)
- Name conventions for methods
  - start with verb and lower case letter
  - examples:
    - printHeader, findMaximum, traverseList, ...



Methods in Java

```
public class SubroutineExample {
    private static void printRule() {  // method head
        System.out.println("-------------"); // method body
    }

    public static void main(String[] args) {
        printRule();                    // method call
        System.out.println("Header 1");
        printRule();
    }

}
```

Parameters

- Input of values supported by methods

```
class Sample {

   static void printMax (int x, int y) {
      if (x > y) Out.print(x); else Out.print(y);
   }

   public static void main (String[] arg) {
      ...
      printMax(100, 2 * i);
   }
}
```

formal parameters
- in the method head
- are the variables in the method body

actual parameters
- in the method call
- can be expressions

Parameters

- Actual parameters are stored in the variables defined by the formal parameters.
- `x = 100; y = 2 * i;`
  - actual parameters need to be type compatible with the formal parameters.

```
class Sample {

  static void printMax (int x, int y) {
    if (x > y) Out.print(x); else Out.print(y);
  }

  public static void main (String[] arg) {
    ...
    printMax(100, 2 * i);
  }
}
```

Functions

- Functions are methods that return a value.

```
class Sample {

   static int max (int x, int y) {
      if (x > y) return x; else return y;
   }

   public static void main (String[] arg) {
      ...
      int result = 3 * max(100, i + j) + 1;
      ...
   }
}
```

- They have a return type, eg. `int` instead of `void`
- They use the `return` keyword to exit
- Can be used in expressions

Functions vs. Procedures

- Functions
  - methods with return values
  - `static int max (int x, int y) {...}`
- Procedures
  - methods without return values
  - `static void printMax (int x, int y) {...}`

Example

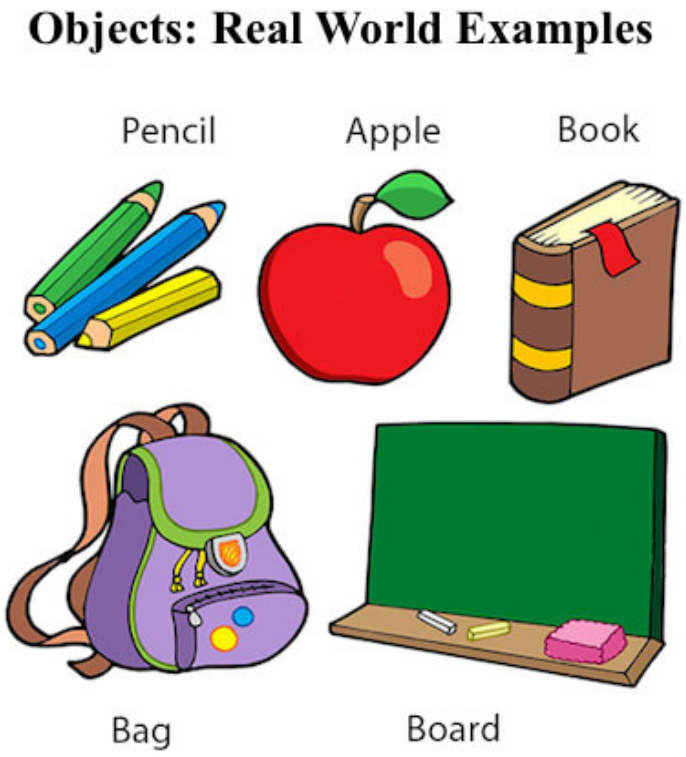
```
public class BinomialCoefficient {
    public static void main(String[] args) {
        int n = 5, k = 3;
        int result = factorial(n) /
                (factorial(k) * factorial(n - k));
        System.out.println("result = " + result);
    }

    public static int factorial(int k) {
        int result = 1;
        for (int i = 2; i <= k; i++) {
            result *= i;
        }
        return result;
    }
}
```

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}.$$

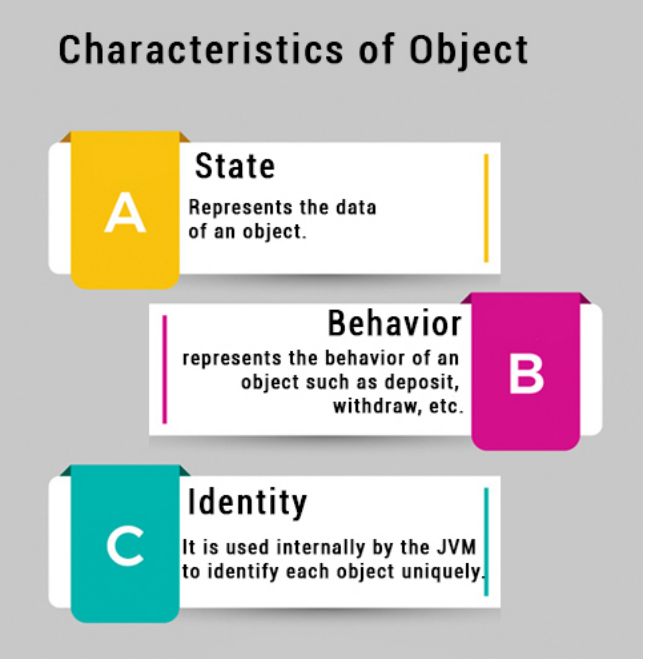

Classes and
Objects

What is actually an object?

- I mean really?



What is an object in Java?



- An entity that has state and behavior
  - e.g. chair, bike, marker, pen, table, car
- An object has three characteristics
  - **State:** represents the data (value) of an object.
  - **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
  - **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.
  - E.g., Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.
- An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

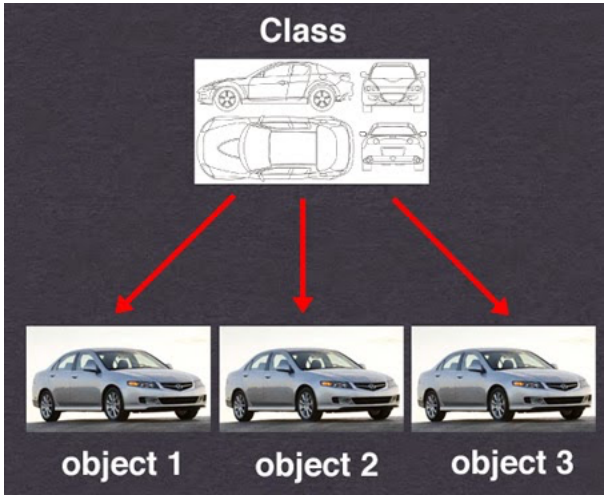
What is a class in Java?

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class can contain: fields, methods, constructors, blocks, nested class and interface
- Syntax to declare a class:

```
class <class_name>{
    field;
    method;
}
```

- Classes have instance variables
  - variable which is created inside the class but outside the method
    - Does not allocate memory at compile time but at run time when instance is created
- Method in the class defines the behavior
  - Code reusability, code optimization
- New keyword is used to allocate memory at runtime.

Objects and classes



- Class is like a template
  - from which instances (objects) are created

- Objects (instances) of a class have to be created explicitely before use.
  - Variable otherwise have the value `null`

```
Car myCar; -> null
Car myCar = new Car(); -> car with inital values
```

Declaration of Classes

Single file

```
class C1 {
   ...
}
class C2 {
   ...
}
class MainProgram {
  public static void
         main (String[] arg) {
         ...
   }
}
```

MainProgram.java

Compile

$> javac MainProgram.java

Multiple files

```
class C1 {
   ...
}
class C2 {
   ...
}
class MainProgram {
  public static void
         main (String[] arg) {
         ...
   }
}
```

C1.java

C2.java

MainProgram.java

Compile

$> javac MainProgram.java C1.java C2.java

Java classes example

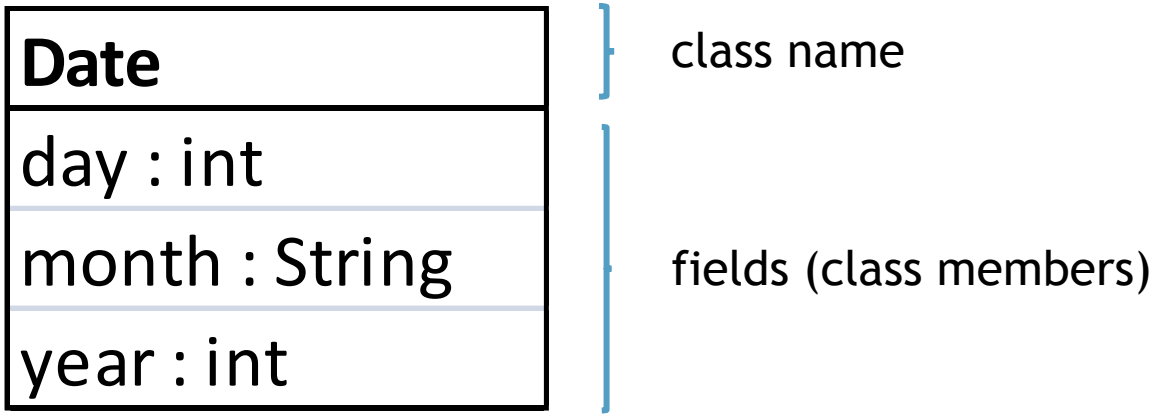
- Example: Store a data in a single structure.
  - day, month, year, ...

- Basic data types not practical for this ...
  - storing more than one
  - return values of functions
  - comparing to other dates

Date
day : int
month : String
year : int

Java classes example

- Combine all necessary variables in one structure:



Data Type Class

- Declaration

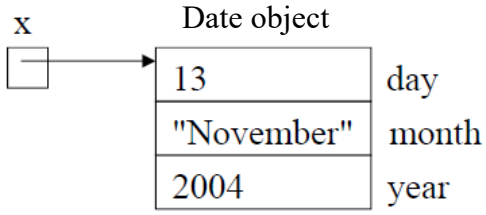
- Data type usage
- Access

```
class Date {
   int day;
   String month;
   int year;
}
```

fields of class date

```
Date x, y;
```

```
x.day = 13;
x.month = "November";
x.year = 2004;
```



Date variables are references / addresses to objects.

Object and class example code

```
//Creating Student class.
class Student{
 int id;
 String name;
}
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }
}
```

3 Ways to initialize objects

Initializing an object means storing data into the object.

By reference variable

By method

By constructor (more details later on this)

Initialization through reference

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Sonoo";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

We can also create multiple objects and store information in it through reference variable.

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Sonoo";
  s2.id=102;
  s2.name="Amit";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

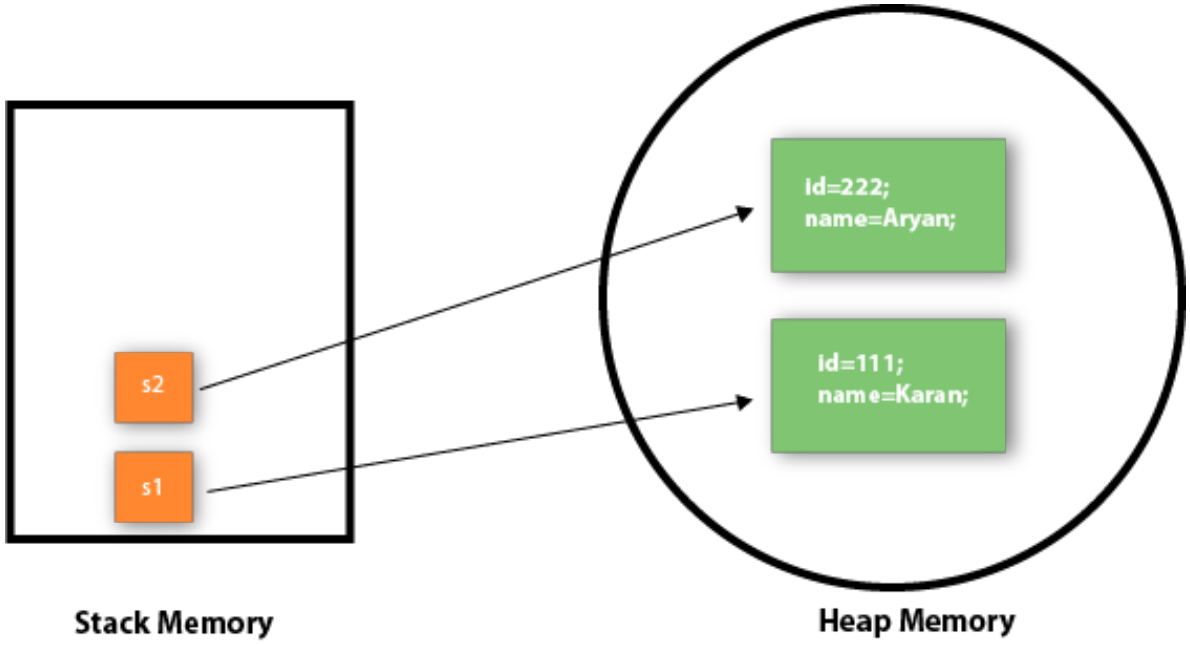
Initialization through method

- Initializing the value to these objects by invoking the insertRecord method

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Karan");
  s2.insertRecord(222,"Aryan");
  s1.displayInformation();
  s2.displayInformation();
 }
}
```

Memory?

- Objects get the memory in heap memory area.
- Reference variables refer to the object allocated in the heap memory area.



Objects intialisation and referencing

```
Date x, y;
```

reserves memory for the address

x,y have value null

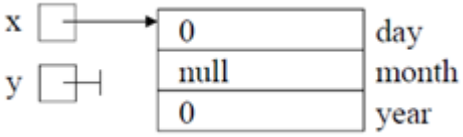
Instantiation

```
x = new Date();
```

creates a new Date object and assigns ist address to x.

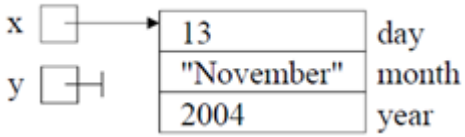
Initial values are

0, null, false or ‚'\u0000'



Usage

```
x.day = 13;
x.month = „November“
x.year = 2004;
```



Assignments

Reference / address assignment

changes x.day too!

Assignments

```
class Date {
   int day;
   String month;
   int year;
}
```

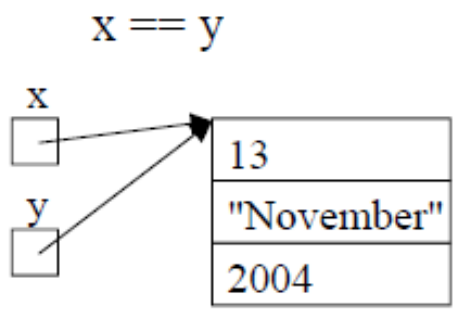
```
class Address {
   int number;
   String street;
   int zipCode;
}
```

```
Date d1, d2 = new Date();
Address a1, a2 = new Address();
```

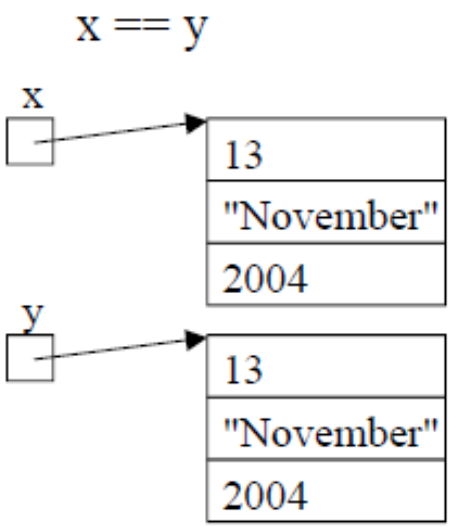
```
d1 = d2;   // ok, same type
a1 = a2;   // ok, same type
d1 = a2;   // not ok, different type (although structure is the same)
```

Comparing references

- `x == y` und `x != y` ... compares references
- `<, <=, >, >=` ... not applicable



x == y returns true



x == y returns false

Compares actual values

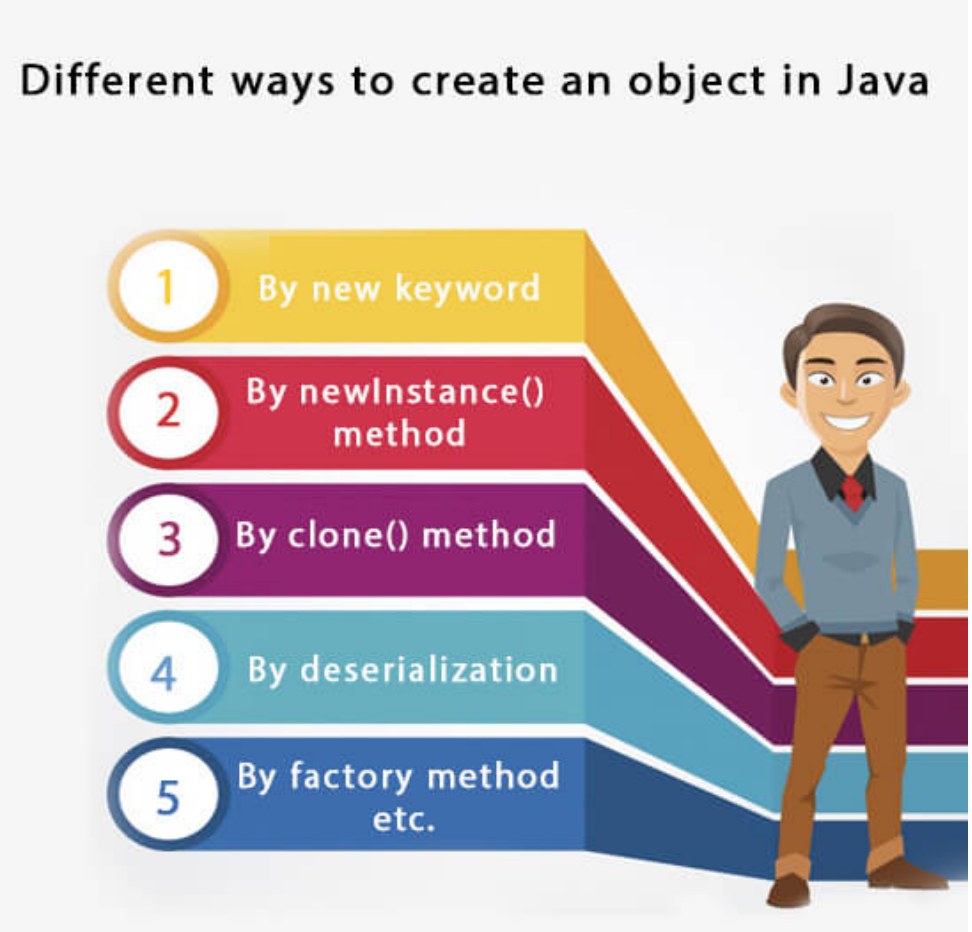
- Has to be implemented by a method.

```
static boolean equalDate (Date x, Date y) {
  return x.day == y.day &&
    x.month.equals(y.month) &&
    x.year == y.year;
}
```

Employee class example

```
class Employee{
    int id;
    String name;
    float salary;
    void insert(int i, String n, float s) {
        id=i;
        name=n;
        salary=s;
    }
    void display(){System.out.println(id+" "+name+" "+salary);}
}
public class TestEmployee {
public static void main(String[] args) {
    Employee e1=new Employee();
    Employee e2=new Employee();
    Employee e3=new Employee();
    e1.insert(101,"ajeet",45000);
    e2.insert(102,"irfan",25000);
    e3.insert(103,"nakul",55000);
    e1.display();
    e2.display();
    e3.display();
}
}
```

What are the different ways to create an object in Java?



- We will learn these ways later…

Anonymous object

- Anonymous simply means nameless.
  - An object which has no reference is known as an anonymous object.
  - It can be used at the time of object creation only.
  - If you have to use an object only once then it might be a clever joice…
  - Usually you would use myObject = new Object() but…

```
class Calculation{
 void fact(int  n){
  int fact=1;
  for(int i=1;i<=n;i++){
   fact=fact*i;
  }
 System.out.println("factorial is "+fact);
}
public static void main(String args[]){
 new Calculation().fact(5);//calling method with anonymous object
}
}
```

Multiple objects by one type only

- We can create multiple objects by one type only as we do in case of primitives.
- Initialization of primitive variables:

int a=10, b=20;

//creating two primitive datatypes

- Initialization of reference variables:

Rectangle r1=**new** Rectangle(), r2=**new** Rectangle();

//creating two objects

What can we do with classes and objects?

- Classes can use other classes
  - and extend on that

```
class Point {
  int x, y;
}

class Polygon {
  Point[] pt;
  int color;
}
```

Has a relation – also called aggregation, later more

What can we do with classes and objects?

- Implement methods with multiple return values

```
class Time {
   int hours, minutes, seconds;
}
class Program {
   static Time convert (int seconds) {
      Time myTime = new Time();
      myTime.hours = seconds / 3600;
      myTime.minutes = (seconds % 3600) / 60;
      myTime.seconds = seconds % 60;
      return time;
   }
   public static void main (String[] arg) {
      Time myTime = convert(10000);
      System.out.println(tmyTime.hour + ":" + myTime.minute + ":" + myTime.seconds);
   }
}
```

What can we do with classes and objects?

- Combination of classes and arrays

```
class Person {
   String name, phoneNumber;
}

class Phonebook {
   Person[] entries;
}

class Program {
   public static void main (String[] arg) {
      Phonebook phonebook = new Phonebook();
      phonebook.entries = new Person[10];
      phonebook.entries[0].name = “John Doe"
      phonebook.entries[0].phoneNumber = "+47 123 456 789"
      // ...
   }
}
```

What can we do with classes and objects?

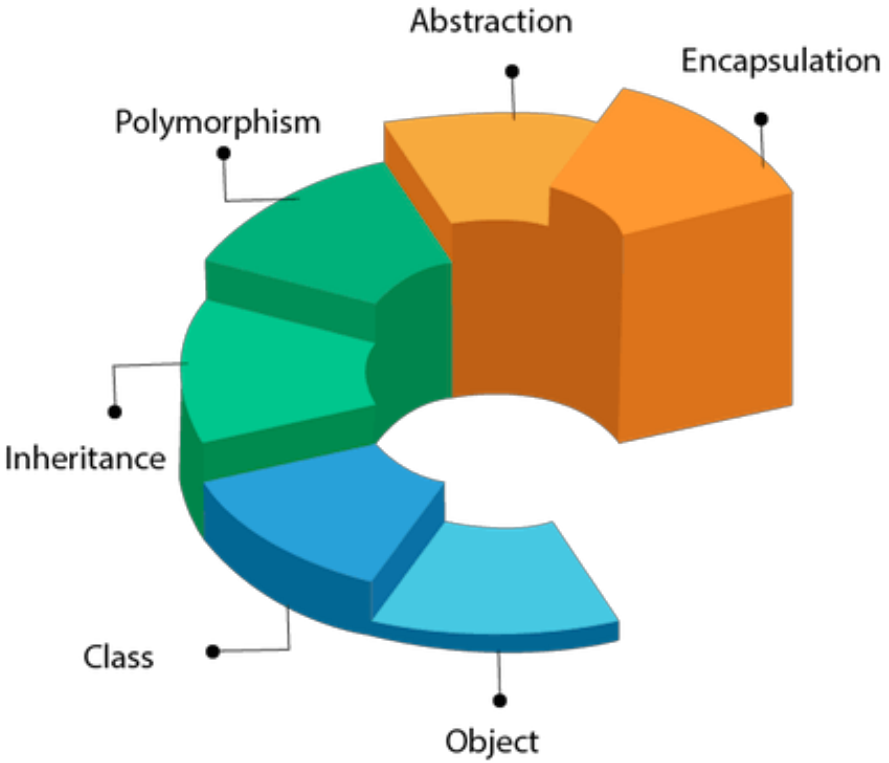
- Combination of classes and arrays

```
class Person {
   String name, phoneNumber;
}

class Phonebook {
   Person[] entries;
}

class Program {
   public static void main (String[] arg) {
      Phonebook phonebook = new Phonebook();
      phonebook.entries = new Person[10];
      phonebook.entries[0].name = “John Doe"
      phonebook.entries[0].phoneNumber = "+47 123 456 789"
      // ...
   }
}
```


Constructors,
static, this

Java OOPs Concepts

- Object-Oriented Programming is a paradigm that provides many concepts such as inheritance, data binding, polymorphism, etc.
  - **Simula** is considered the first object-oriented programming language
  - **Smalltalk** is considered the first truly object-oriented programming language
- Popular object-oriented languages are Java, C#, PHP, Python, C++, etc.
- The main aim of object-oriented programming is to implement real-world entities for example object, classes, abstraction, inheritance, polymorphism, etc.



Java OOPs Concepts

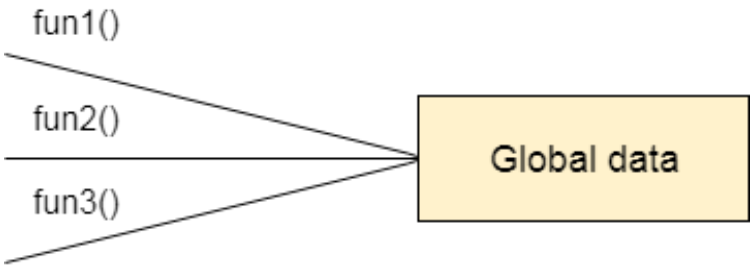
- Class: A collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class does not consume any space on the memory.
- Object: Is an instance of a class. Takes space on the memory and can communicate with other objects knowing the details of each others data or code.
- Inheritance: When one object acquires all the properties and behaviors of a parent object, it is known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.
- Polymorphism: Simply said the same task is performed in different ways. In Java, we use method overloading and method overriding to achieve polymorphism. E.g., animals speak different

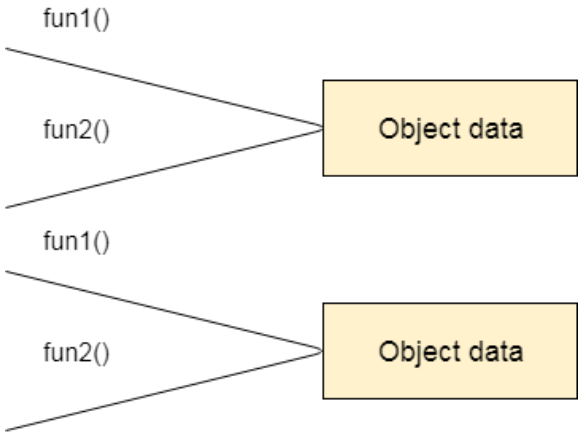
Java OOPs Concepts

- Abstraction: Hiding internal details and showing functionality is known as abstraction. For example for a phone call, we do not know the internal processing (what is actually happening and we do not care).
  - In Java, we use abstract class and interface to achieve abstraction.
- Binding (or wrapping) code and data together into a single unit are known as encapsulation. Can be imagined as a capsule wrapping different medicine.
  - A java class is the example of encapsulation.

Advantage of OOPs over procedure-oriented programming language

- OOP makes development and maintenance easier whereas in a procedure-oriented programming language it is not easy to manage if code grows as project size increases.
- OOP provides data hiding whereas in a procedure-oriented programming language global data can be accessed from anywhere.
- OOP provides the ability to simulate a real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.





Constructors

- A constructor is a block of code similar to a method. It is called when an instance of the object is created, and memory is allocated for the object.
- Basically a special type of method which is used to initialize the object.

When is a constructor called?

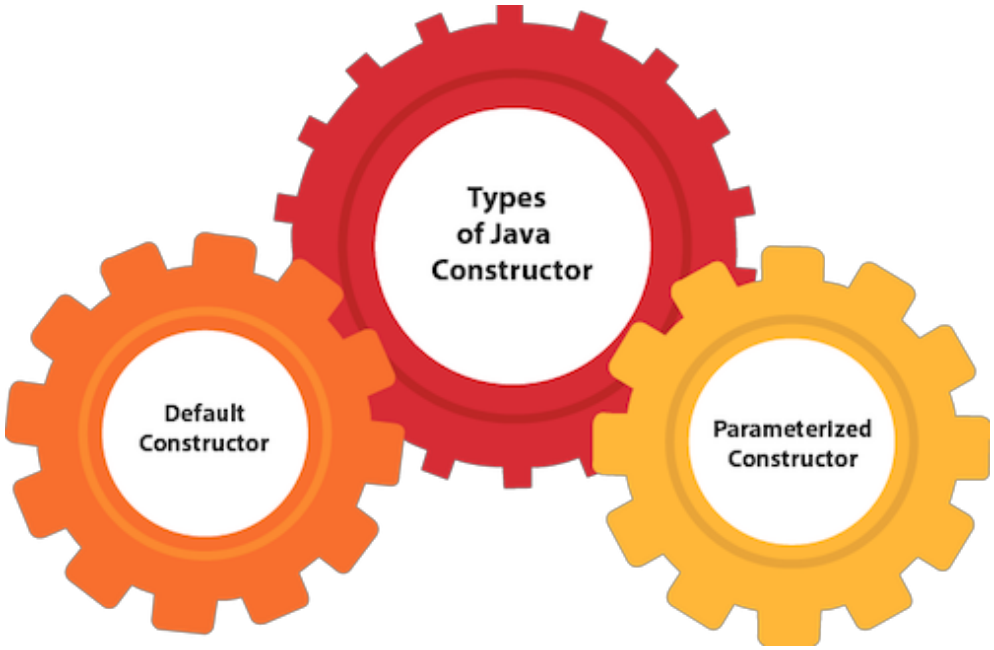
- Every time an object is created using the new() keyword, at least one constructor is called.
  - It calls a default constructor if not specified.
- **Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class (Default constructor).**

Rules for creating a constructor

- There are three rules defined for the constructor:
  - A constructor name must be the same as its class name
  - A constructor must have no explicit return type
  - A constructor cannot be abstract, static, final or synchronized
- **Note: We can use access modifiers while declaring a constructor. It controls the object creation.**
  - In other words, we can have private, protected, public or default constructors in Java.

Types of constructors

- There are two types of constructors in Java
- 1) Default constructor (no-arguments constructor)
- 2) Parameterized constructor



Java Default Constructor

- A constructor is called "Default Constructor" when it does not have any parameters.

<class_name>(){}

- In the following example, we are creating the no-arguments constructor for a bike class. It is called at the time of object construction/creation (bike1).
- **Rule: If there is no constructor in a class, the compiler automatically creates a default constructor.**

Java default constructor example

- Bike1

```
//Java Program to create and call a default constructor
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

What is the purpose of the default constructor?

- The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.
  - Remember we talked about this before!

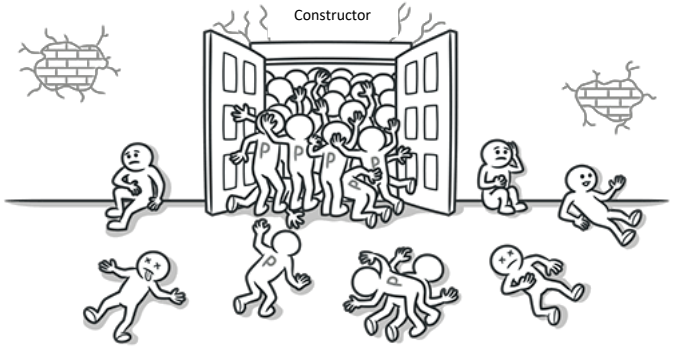
Example of default constructor that displays the default values

- In the example the compiler provides a default constructor. The default constructor provides 0 and null values (Student1).

```
//Let us see another example of default constructor
//which displays the default values
class Student3{
int id;
String name;
//method to display the value of id and name
void display(){System.out.println(id+" "+name);}

public static void main(String args[]){
//creating objects
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();
s2.display();
}
}
```

Java parameterized constructor

- Is a constructor which has a specific number of parameters.
- The parameterized constructor is used to provide different values to the distinct objects. However, nothing prevents from providing the same values (will be a new object anyway).
- The next example, we create the constructor for the Student class that has two parameters. There is no limitation for the number or parameters (Student2)!



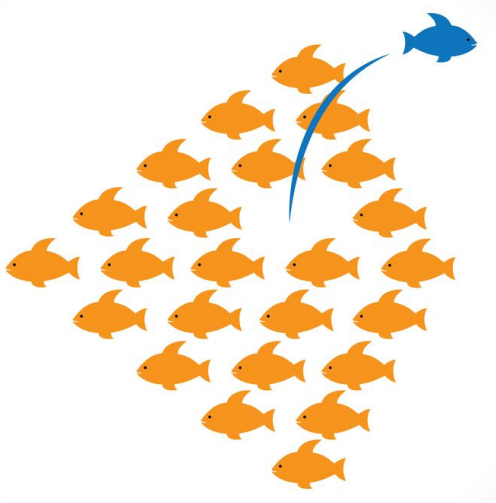
Example of a parameterized constructor

- Student2

```
//Java Program to demonstrate the use of parameterized constructor
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Constructor overloading

- In Java, a constructor is just like a method but without a return type. It can also be overloaded like methods.
- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.
- They are differentiated by the compiler by the number of parameters in the list and their types.



Example of constructor overloading

- Student3

```
//Java program to overload constructors in java
class Student5{
   int id;
   String name;
   int age;
   //creating two arg constructor
   Student5(int i,String n){
   id = i;
   name = n;
   }
   //creating three arg constructor
   Student5(int i,String n,int a){
   id = i;
   name = n;
   age=a;
   }
   void display(){System.out.println(id+" "+name+" "+age);}

   public static void main(String args[]){
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
  }
}
```

Difference between constructors and methods in Java

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as class name.

Copy constructor

- In C++ we can copy the values from one object to another with copy constructor.
- There is no copy constructor in java.
- Although there are many other ways to copy the values of one object into another in java.
  - With a constructor
  - By assigning the values of one object into another
  - By the clone() method of the Object class
- In the next example we are going to copy the values of one object into another using a constructor.

Copy constructor example

- Student4

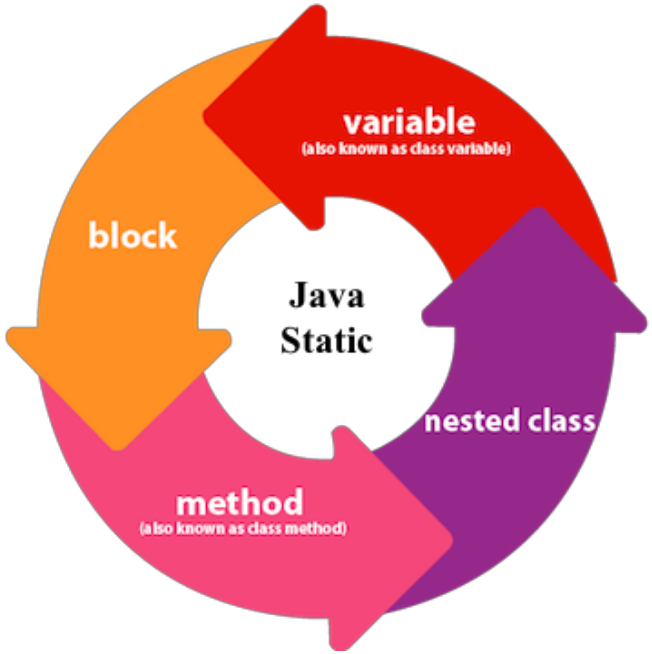
```
//Java program to initialize the values from one object to another
class Student6{
    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i,String n){
    id = i;
    name = n;
    }
    //constructor to initialize another object
    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

Copying values without constructor

- We can also copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor (Student5).
- Q) Does constructor return any value?
  - Yes, it is the current class instance (cannot use return type yet it returns a value).
- Q) Can constructor perform other tasks instead of initialization?
  - Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform it in the method.

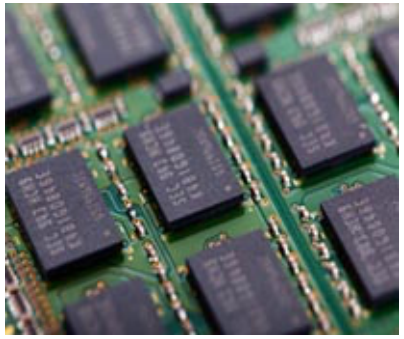
Java static keyword

- The **static keyword** in Java is mainly used for memory management. We can apply java static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class and not to an instance of the class.
- Static can be:
  - A variable (also known as a class variable)
  - A method (also known as a class method)
  - A block
  - A nested class



Java static variable

- If you declare any variable as static, it is known as a static variable.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object).
  - For example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.
- Advantages of static variable
  - It makes your program **memory efficient** (i.e., it saves memory).



Understanding the problem without static variable

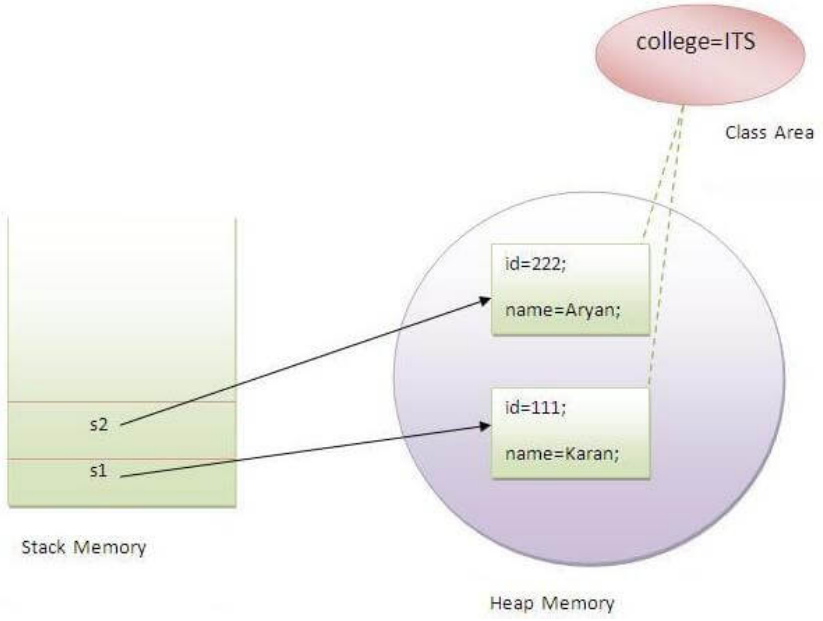
- We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create a constructor.
- Suppose there are 500 students in a college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name. Instance data members are necessary for these two.
  - But "college" refers to the common property of all objects. If we make it static, this field will get memory assigned only once.
- Java static property is shared to all objects.

```
class Student{
    int rollno;
    String name;
    String college="ITS";
}
```

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student{
   int rollno;//instance variable
   String name;
   static String college ="ITS";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

- TestStaticVariable1



Program of a counter without static variable

- In this example, we have created an instance variable named count which is incremented in the constructor. Since the instance variable gets the memory at the time of object creation, each object will have a copy of the instance variable.
- If it is incremented, it will not reflect other objects. Therefore, each object would have the value 1 in the count variable (Counter).

Program of a counter using static variable

- As mentioned before, a static variable will get the memory assigned only once.
- If any object changes the value of the static variable, it will retain its value.
- Example Counter2

Static method

- The static keyword applied with any method is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access a static data member and can change the value of it.
- Two examples: students and a static method that performs a normal calculation (TestStaticMehtod and Calculate)

Restrictions for the static method

- There are two main restrictions for the static method.
  - The static method cannot use non static data members or call non-static methods directly (Example A).
  - Keywords this and super cannot be used in static context.

- Why is the Java main method static?
- Because an object is not required to call a static method. If it would be a non-static method, JVM creates an object first then calls the main() method which would lead to the problem of extra memory allocation.

Static block

- Is used to initialize the static data members.
- It is executed before the main method at the time of class loading (Example A2).

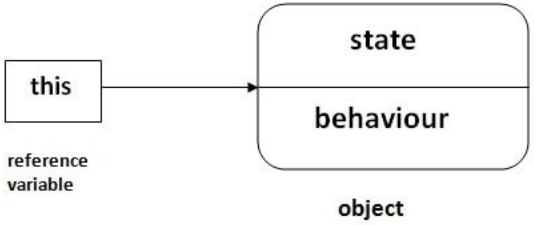
- Can we execute a program without main() method?
- No, one of the ways was the static block, but it was only possible until JDK 1.6. Since JDK 1.7, it is not possible to execute a java class without the main method (Example A3).

```
class A3{
  static{
  System.out.println("static block is invoked");
  System.exit(0);
  }
}
```

Will not work!

this keyword in java

- The this keyword can be used in several different ways. In Java, **this** is a **reference variable** that refers to the current object.



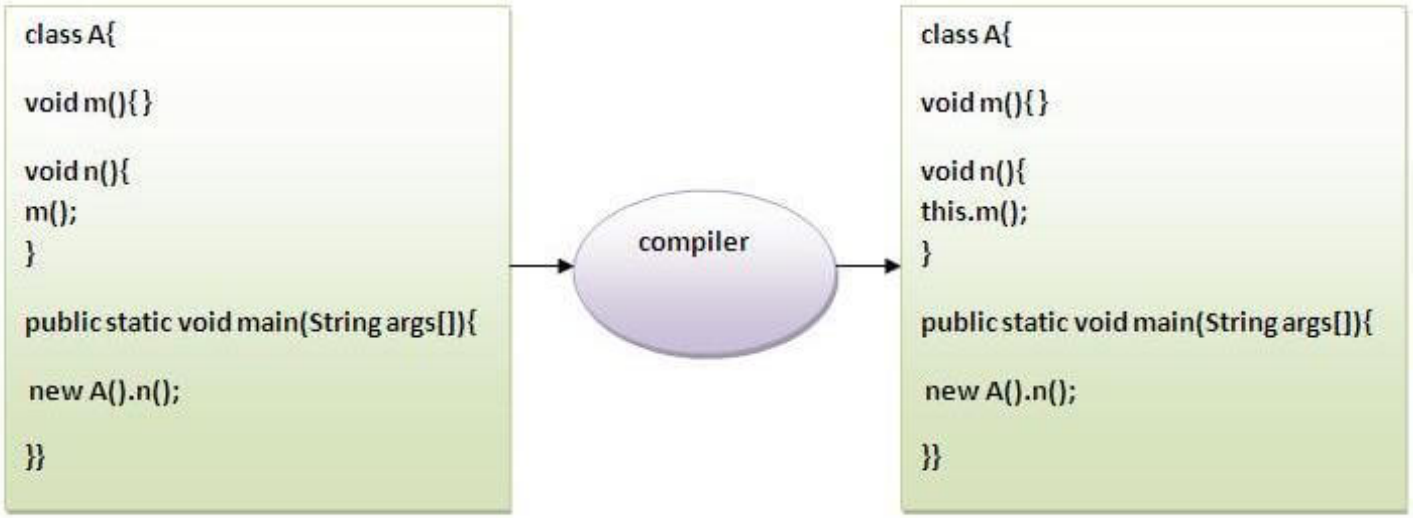
1) this: to refer to a current class instance variable

- The this keyword can be used to refer to a current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.
- Understanding the problem without this keyword
- Let's understand the problem if we don't use this keyword by the example (TestThis1)
- In the example, parameters (formal arguments) and instance variables are the same. We are not using the this keyword to distinguish local variables and instance variables.
- Solution of the above problem by the this keyword (TestThis2)
- If the local variables (formal arguments) and instance variables are different, there is no need to use the this keyword like in the next example (TestThis3, program where this keyword is not required)

It is better approach to use meaningful names for variables. So we use same name for instance variables and parameters in real time, and always use this keyword.

2) this: to invoke a current class method

- A method of the current class can be invoked by using the this keyword.
- If you do not use the this keyword, the compiler automatically adds this keyword while invoking the method. Example **TestThis4** displays this behavior.



3) this: to invoke current class constructor

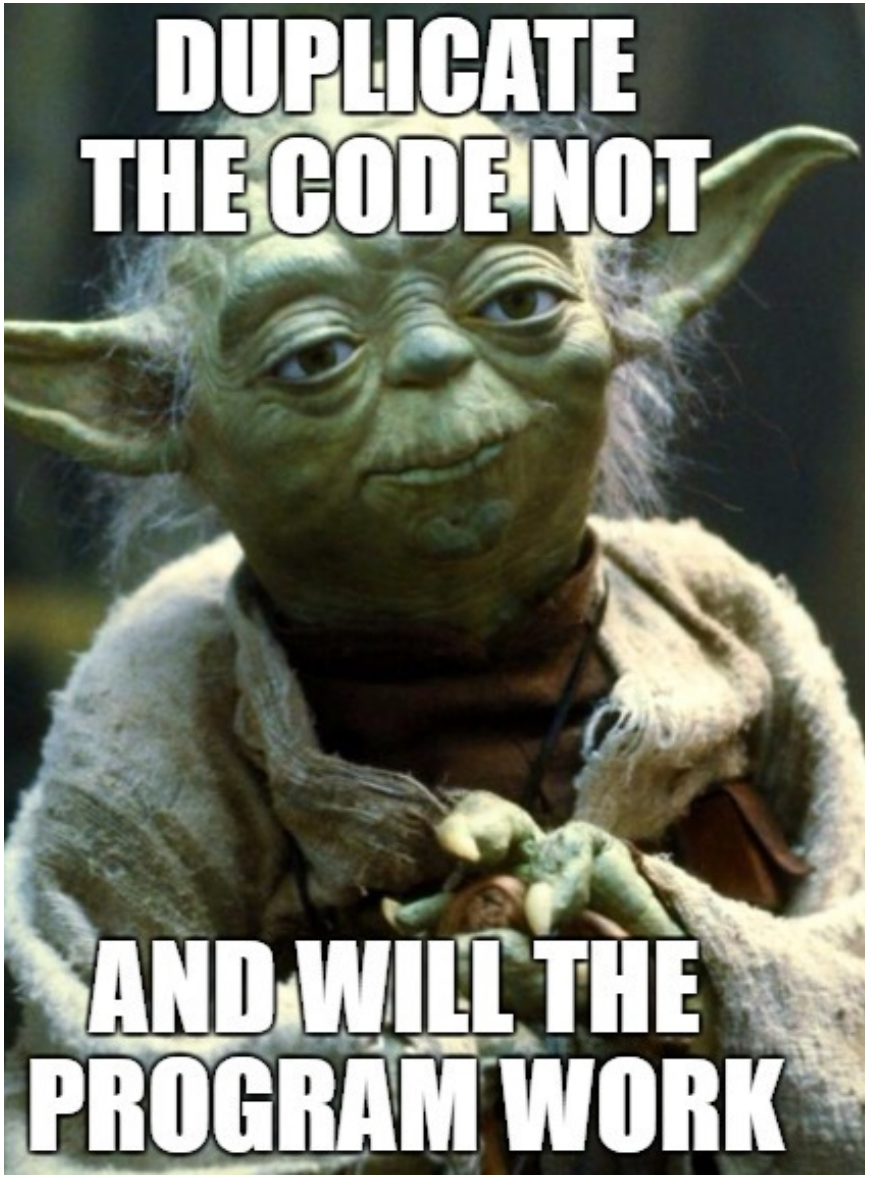
- The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.
- Example 1: **Calling the default constructor from parameterized constructor (TestThis5)**
- Example 2: **Calling a parameterized constructor from default constructor (TestThis6)**
- Example of real world usage of this() constructor call
- The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining (TestThis7).
- **Rule: Call to this() must be the first statement in constructor (TestThis8)**

4) this: to pass as an argument in the method

- The this keyword can also be passed as an argument in the method. It is mainly used in the event handling (S2).
- Application of this that can be passed as an argument:
- In event handling (or) in a situation where we have to provide a reference of a class to another one. It is used to reuse one object in many methods.

5) this: to pass as argument in the constructor call

- We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example (A4)



6) this keyword can be used to return the current class instance

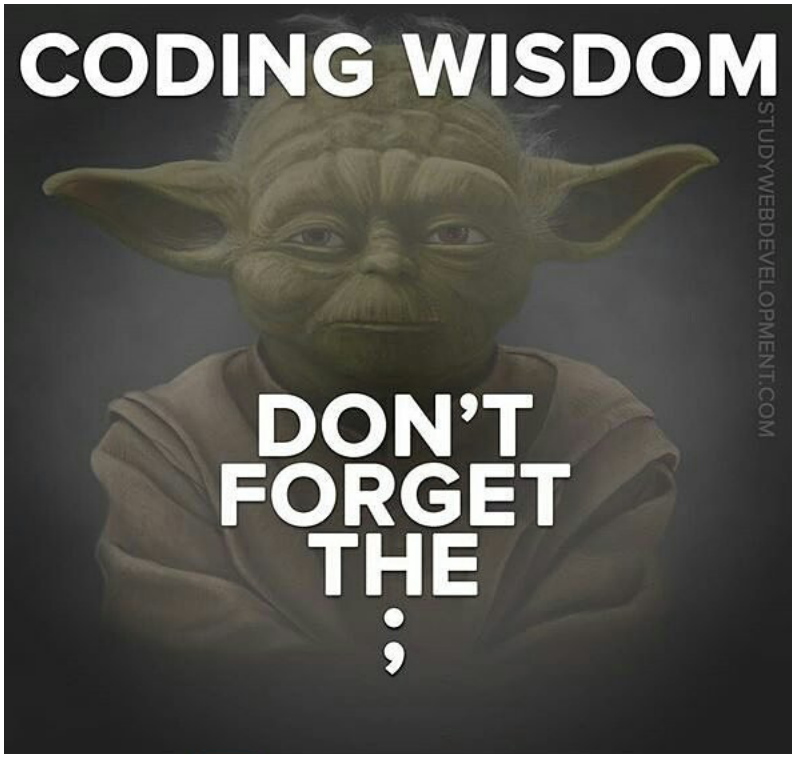
- We can return the this keyword as an statement from the method. In such case, the return type of the method must be the class type (non-primitive).
- Syntax of this that can be returned as a statement

```
return_type method_name(){
return this;
}
```

Example of this keyword that you return as a statement from the method (Test1)

Proving this keyword

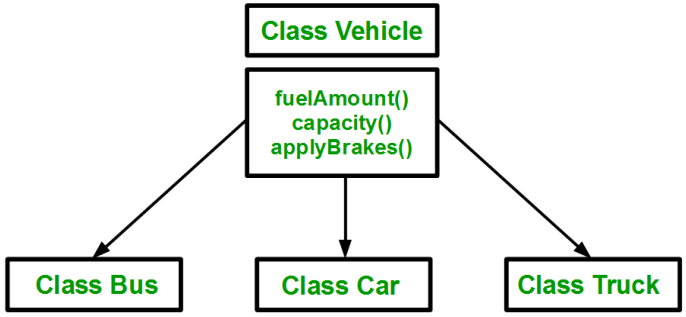
- Let's prove that the this keyword refers to the current class instance variable. In the next example program, we are printing the reference variable and this, output of both variables should be the same (A5).




Inheritance,
aggregation,
composition, super, final

Inheritance

- Is a mechanism in which **one object acquires all the properties and behaviors of a parent object** (important part of OOP).
- The idea behind inheritance in Java is that you can create **new classes** that are **built upon existing classes**.
- When you inherit from an existing class, you can **reuse methods and fields** of the parent class.
  - You can also add new fields and methods in your current class
- Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship.
- Why use inheritance?
  - For method overriding (so runtime polymorphism can be achieved).
  - For code reusability.



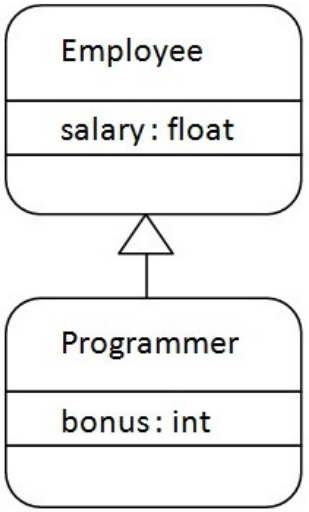
Terms used for inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which **inherits** the **other class**. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class **from where a subclass inherits the features**. It is also called a base class or a parent class.
- **Reusability:** Reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Syntax of inheritance

- The **extends** keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to extend the functionality of the parent class.
- In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

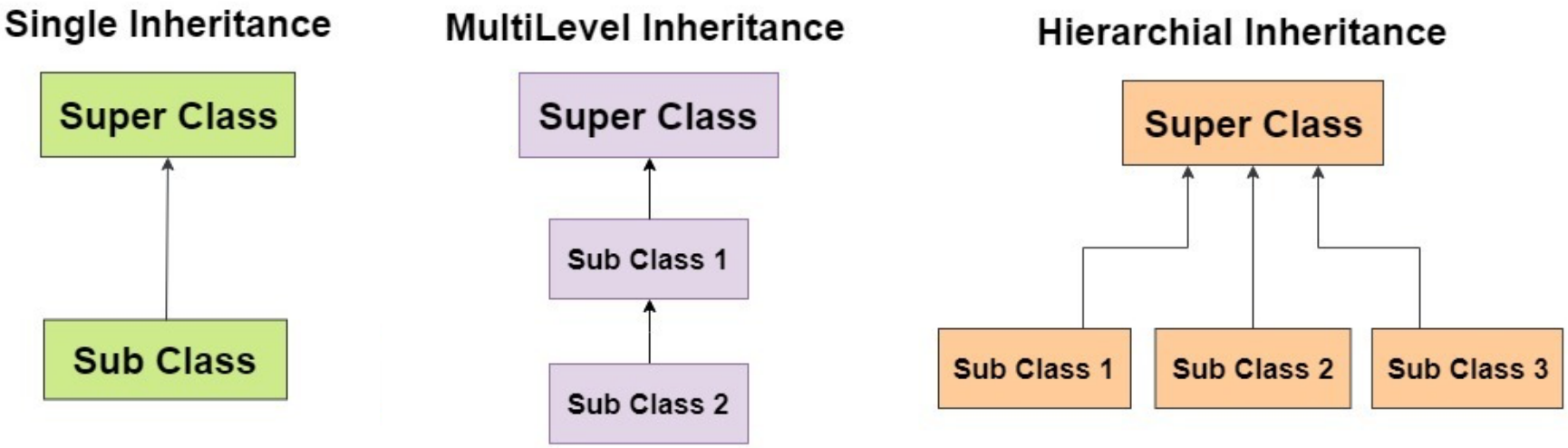
```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```



- Example time: **A programmer is the subclass and Employee is the superclass**. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee (Employee.java).

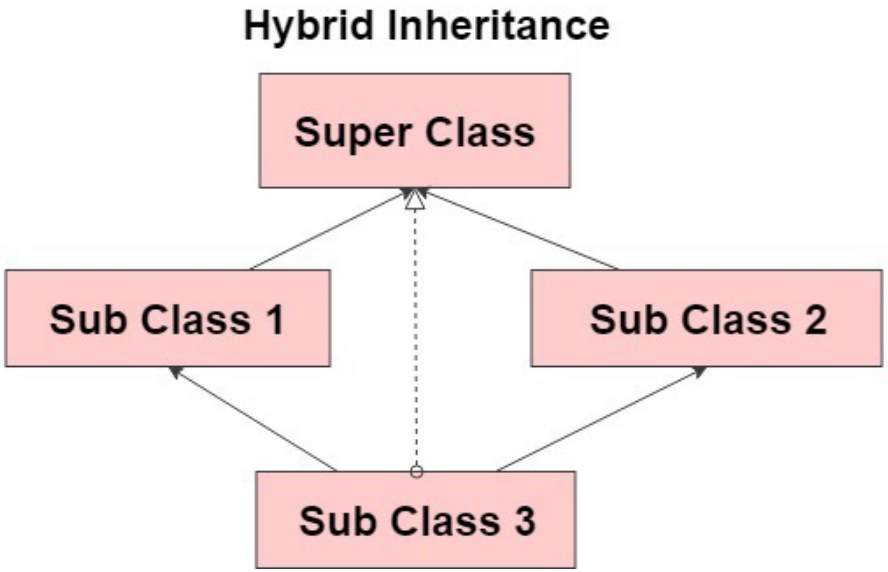
Types of inheritance (i)

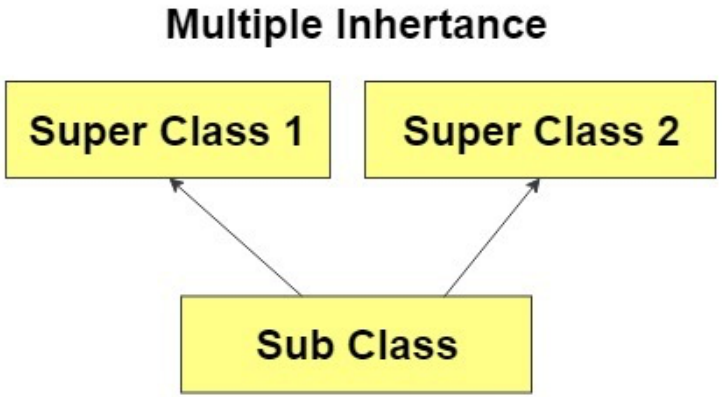
- On the basis of class, there can be three types of inheritance:
- **single, multilevel and hierarchical**.



Types of inheritance (ii)

- When one class inherits multiple classes, it is known as multiple inheritance.
- In Java, multiple and hybrid inheritance is supported through interfaces (we will learn about them later!).





Why multiple inheritance is not supported in java

- Main reason: reduce the complexity and simplify the language.
- **Why?**
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have the same method or different, there will be a compile time error (CompilTimeError.java).

Inhertiance examples

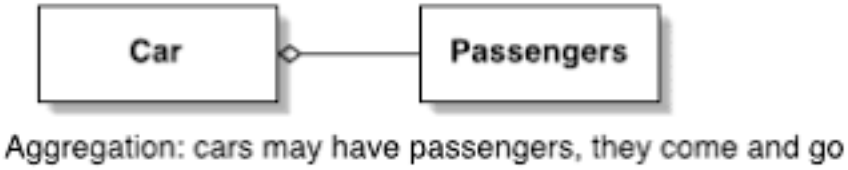
- Single inheritance example (TestInheritance.java)
- Multilevel inheritance example (TestMultilevelInheritance.java)
- Hierarchical inheritance example (TestHierarchicalInheritance.java)

Aggregation

- Aggregation represents a **HAS-A relationship**.

- Example: An Employee object contains information such as id, name, emailId, etc. It contains one more object named address, which contains its own information such as city, state, country, zipcode etc.

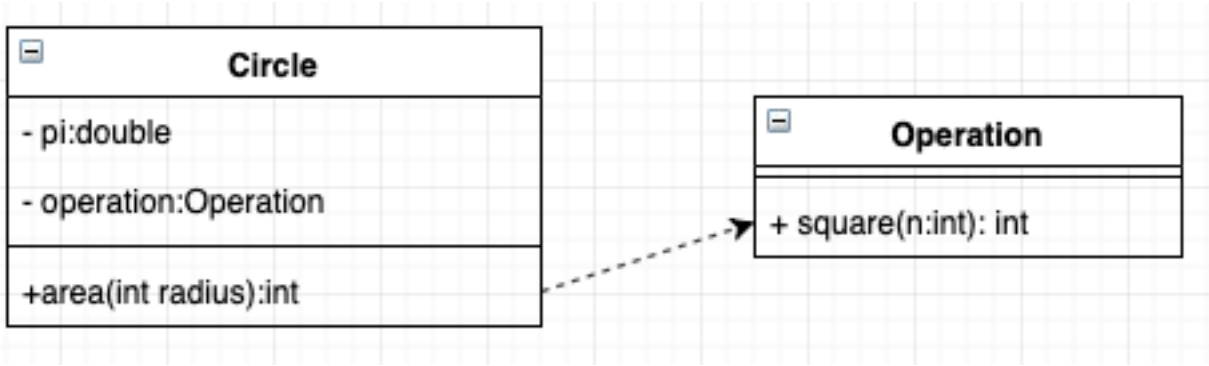
```
class Employee{
int id;
String name;
Address address;//Address is a class
...
}
```



Aggregation: cars may have passengers, they come and go

- Employee has an entity reference address, so the relationship is Employee HAS-A address.

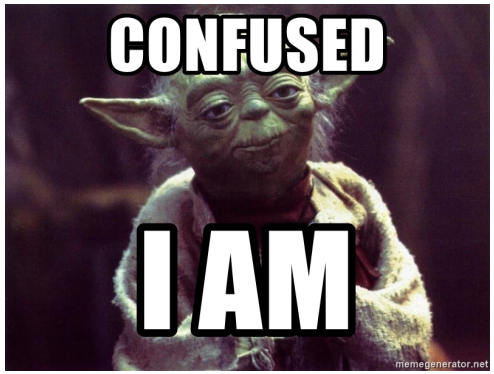
Aggregation example



- com.pgr103.aggregation

When to use aggregation?

- Code reuse most efficient by aggregation when there is **no is-a** relationship.
- If the **is-a relationship** is maintained throughout the lifetime of the objects involved, then inheritance is the way to go: otherwise, aggregation is the best choice.
- Meaningful example of aggregation
  - Employee has an object of type Address. The address object contains its own information such as city, state, country etc. Relationship: Employee HAS-A address (com.pgr103.com.pgr103.aggregation.example2).



Composition

Composition: every car has an engine.

Aggregation: cars may have passengers, they come and go

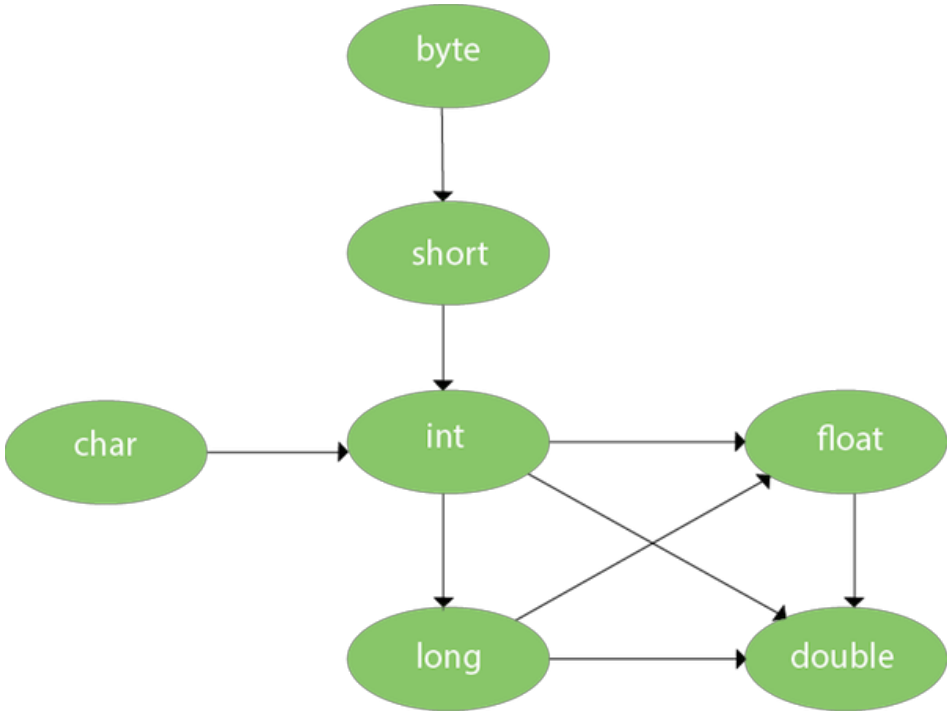
- Composition is a special case of aggregation.
  - Restricted aggregation
- When an object contains the other object and if the contained object **cannot exist without the existence of container object**, then it is called composition.
- Example: A human contains a heart. A human cannot exist without a heart. There exists composition between human and heart.
- Composition is basically a design technique to implement has-a relationships in classes. We can use inheritance or object composition for code reuse.
- Java composition is achieved by using instance variables that refers to other objects (com.pgr103.composition example).

Method overloading - Repetition

- If a class has multiple methods having the same name but they are different in their parameters, it is known as **Method Overloading**.
  - Increase readability of our program
- Two different ways:
  - By changing number of arguments (com.pgr103.overloading.adder.java)
  - By changing the data type (com.pgr103.overloading.adder2.java)
  - Method Overloading is not possible by only changing the return type of the method (**because of ambiguity**, com.pgr103.overloading.adder3.java).
- You could also overload the main method! You can have any number of main methods in a class by method overloading. But JVM calls the main() method which only receives a string array as arguments. Others are ignored.

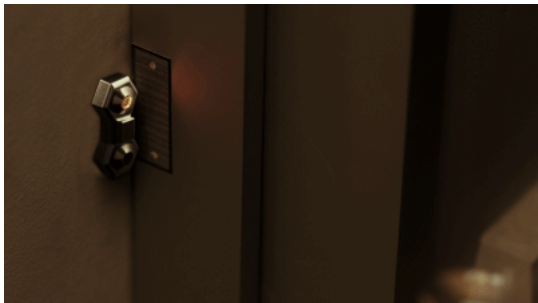
Method overloading and type promotion

- One type is promoted to another implicitly if no matching datatype is found (OverloadingCalculation1, 2 and 3 examples).
  - Byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int, long, float or double and so on.



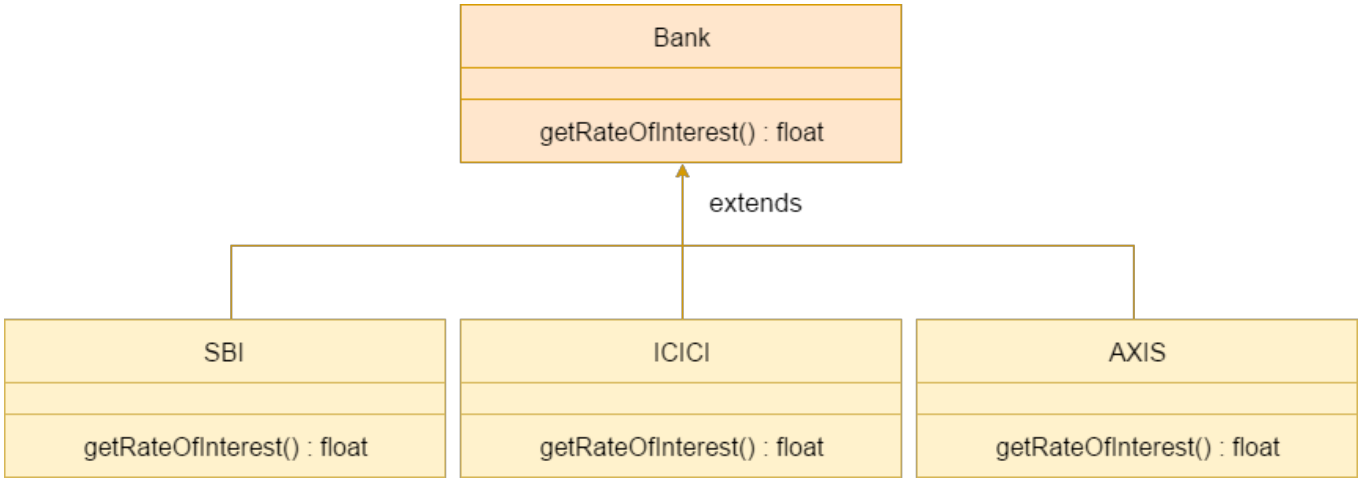
Method overriding

- If a subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
  - The **subclass provides the specific implementation of the method** that has been declared by one of its parent classes
- Usage of method overriding
  - Method overriding is used to **provide the specific implementation of a method** which is **already provided by its superclass**.
  - Method overriding is used for runtime polymorphism
- Rules
  - The method must have the **same name** as in the **parent** class
  - The method must have the **same parameter** as in the parent class.
  - There must be an **IS-A relationship** (inheritance).



Method overriding examples

- No method overriding (com.pgr103.overriding.bike)
- With method overriding (com.pgr103.overriding.bike)
- A real world example of method overriding
  - Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to the banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest (com.pgr103.overriding.BankRateExample).



Can we override static methods?

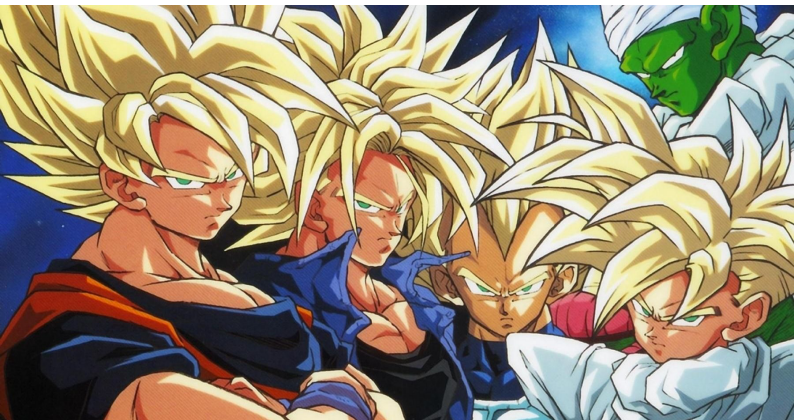
- No, a static method cannot be overridden.
- The reason is that the **static method is bound with the class** whereas the **instance method is bound with an object**. Static belongs to the class area, and an instance belongs to the object area.
- The main method can also not be overridden because it is a static method.

Differences between method overloading and overriding

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Super keyword

- The **super** keyword in Java is a **reference variable** which is used to refer immediate parent class object.
- Whenever an **instance of subclass** is created, an **instance of the parent class is created** implicitly which is referred by the super reference variable.
- Usage of super
  - super can be used to refer to an **immediate parent class instance variable**.
  - super can be used to invoke an **immediate parent class method**.
  - super() can be used to invoke an **immediate parent class constructor**.

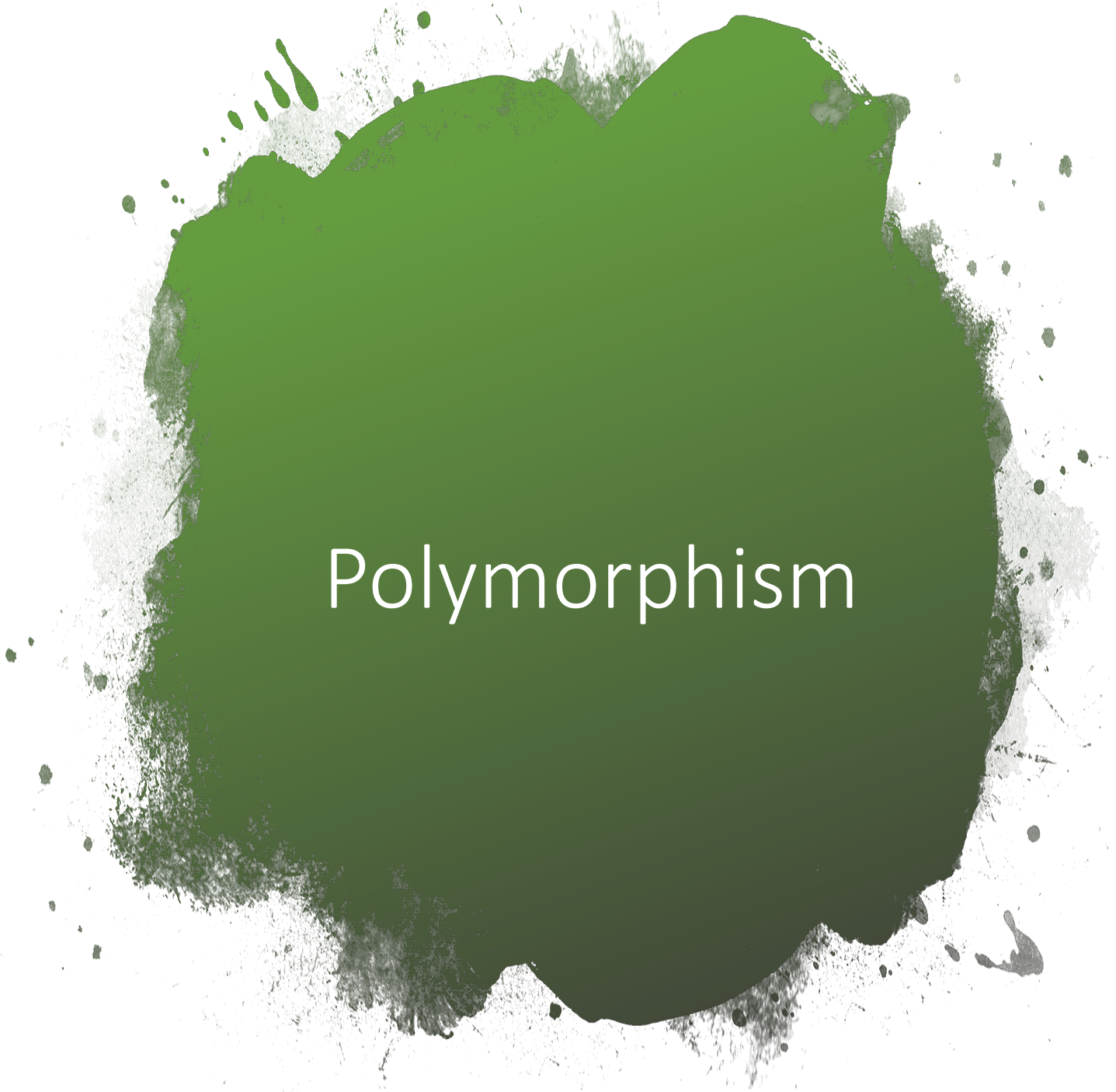


Super examples

- 1) super is used to refer to an immediate parent class instance variable (com.pgr103.superexamples.TestSuper1).
- 2) super can be used to invoke the parent class method (com.pgr103.superexamples.TestSuper2).
- 3) super is used to invoke the parent class constructor (com.pgr103.superexamples.TestSuper3).
- Note: super() is added in each class constructor automatically by the compiler if there is no super() or this().
- super example and a real use case
  - Employee class inherits Person class so all the properties of Person will be inherited to Employee by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor (com.pgr103.superexamples.SuperEmployee).

Final keyword

- The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context.
- Any **variable declared as final cannot be changed** (It will be constant).
- Final can be:
  - 1) **variable** – cannot be changed (com.pgr103.finalexamples.Bike)
  - 2) **method** – cannot be overridden (com.pgr103.finalexamples.Bike2)
  - 3) **Class** – cannot be extended (com.pgr103.finalexamples.Bike3)
- A **final method is inherited** but **cannot** be **overridden**! (com.pgr103.finalexamples.Bike4)
- A final variable that is not initialized at the time of declaration is known as blank final variable.
  - If you want to create a variable that is initialized at the time of creating an object and once initialized may not be changed, it is useful. For example speed limit of your bike. It can be initialized in the constructor (com.pgr103.finalexamples.Bike5)


Polymorphism

Repetition: ArrayList

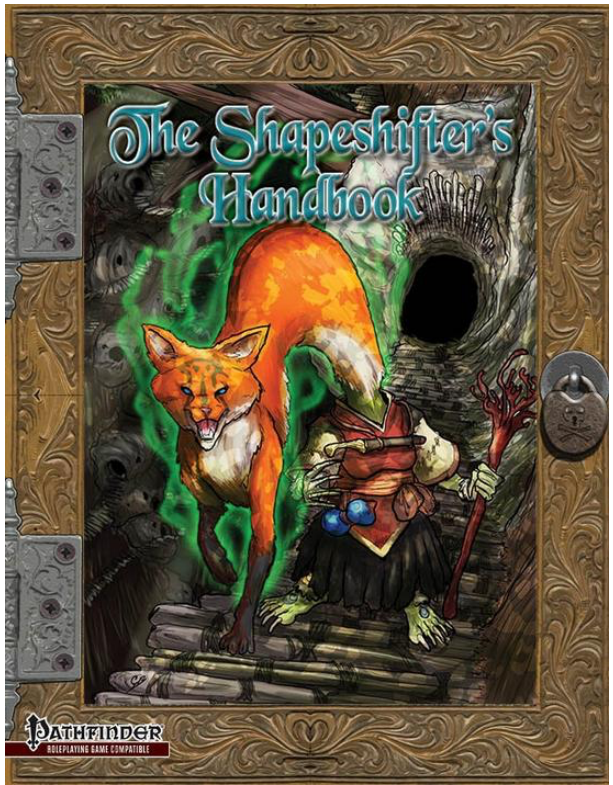
- The ArrayList class is a resizable array, which can be found in the java.util package.
- The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified
  - If you want to add or remove elements to/from an array, you have to create a new one.
- While elements can be added and removed from an ArrayList whenever you want. The syntax is also slightly different:
- Most important methods: add(), get(), set(), remove()

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```

- Example: ArrayListTest

Polymorphism

- **Polymorphism** is a concept by which we can perform a *single action in different ways*. Polymorphism is derived from two Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.



Polymorphism in Java

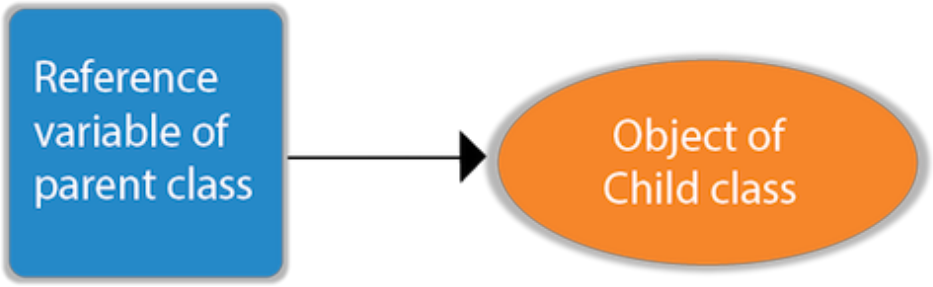
- **In Java there are two types of polymorphism**
- Compile-time polymorphism
  - Resolved at compile time.
  - Overloading a static method is an example of compile time polymorphism.
- Runtime polymorphism
  - Function call will resolved at runtime.
- Polymorphism can be implemented by method overloading and method overriding.

Runtime Polymorphism

- **Runtime polymorphism** (also called **Dynamic Method Dispatch)** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.
- First: understand upcasting before runtime polymorphism

Upcasting

- If the reference variable of a parent class refers to the object of child class, it is known as upcasting.




```
class A{}
class B extends A{}
A a=new B();//upcasting
```

```
interface I{}
class A{}
class B extends A implements I{}
```

```
B IS-A A
B IS-A I
B IS-A Object
```

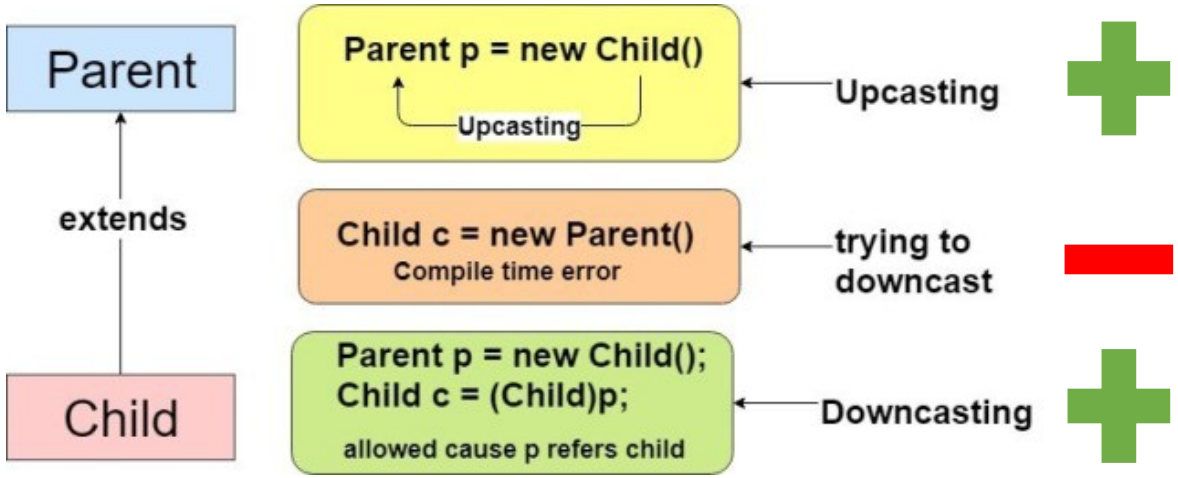
- For upcasting, we can use the reference variable of class type or an interface type.
- Why is B an Object?
  - Since Object is the root class of all classes in Java, so we can write B IS-A Object.

Downcasting

- If the Subclass type refers to the reference, not object, of the parent class, it is known as downcasting.

```
Animal a  = new Dog3();
Dog3 d = (Dog3)a; //Dog 'd' is referring to Animal reference
```

- That means we can downcast only the reference of the Parent class and not an object.



Examples of runtime polymorphism

- An example of two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of the parent class (com.pgr103.polymorphism.Bike).
  - Since it refers to the subclass object and subclass method overrides the parent class method, the subclass method is invoked at runtime.
  - Since method invocation is determined by the JVM not the compiler, it is known as runtime polymorphism.
- Bank example (again)
  - This example was also presented in method overriding but there was no upcasting used.
  - A Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest (com.pgr103.polymorphism.TestPolymorphism).

More examples of runtime polymorphism

- Shape: A program that allows to “draw” different shapes. The shape class is the parent class implementing a generic draw method. The child classes are implementing the specific drawing methods.
- (com.pgr103.polymorphism.TestPolymorphism2)
- Animal: A program that allows to different types of animals from the parent class Animal. The child classes are implementing the specific eat methods.
- (com.pgr103.polymorphism.TestPolymorphism3)

Java Runtime Polymorphism with Data Members

- A method is overridden, not the data members, so runtime polymorphism cannot be achieved by data members.
- What does that exactly mean?
- Example: Two classes have a data member speedlimit. We are accessing the data member by the reference variable of the parent class which refers to the subclass object. Since we are accessing the data member which is not overridden it will always access the data member of the parent class (com.pgr103.polymorphism.Bike2).
- **Lesson learned: Runtime polymorphism cannot be achieved by data members.**

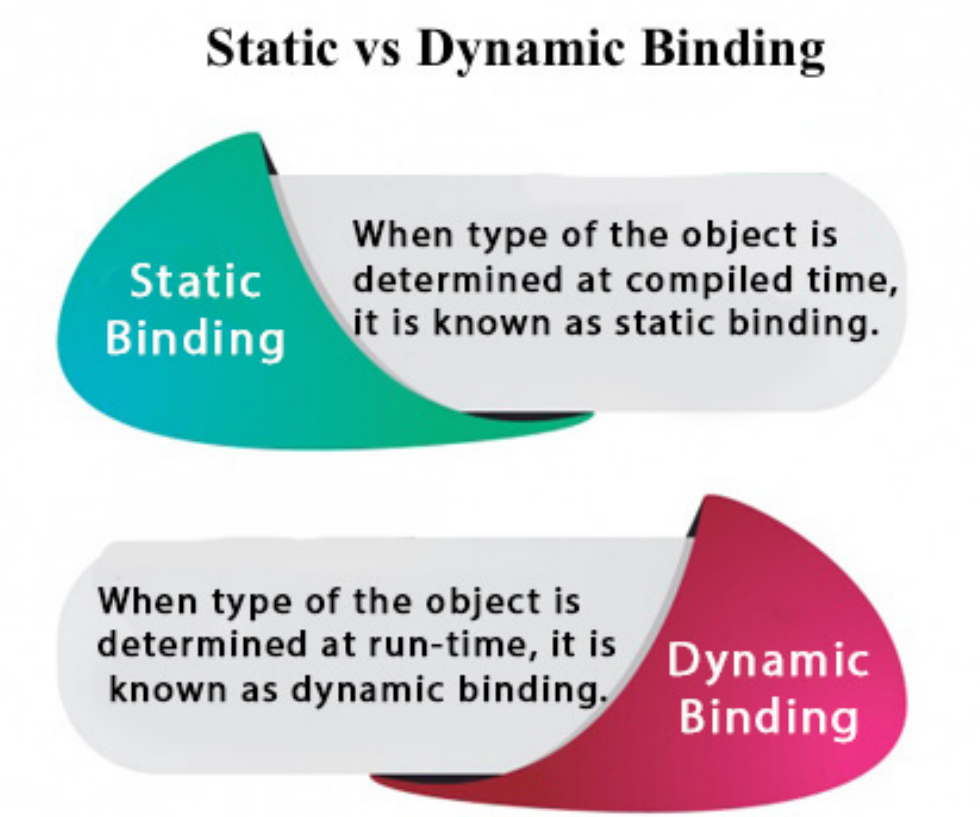
Java Runtime Polymorphism with Multilevel Inheritance

- Example of Runtime Polymorphism with multilevel inheritance (com.pgr103.polymorphism.Animal2).

- What do we get for this code: com.pgr103.polymorphism.Animal3
- BabyDog is not overriding the eat() method, therefore the eat() method of Dog class is invoked.

Static binding and dynamic binding

- Connecting a method call to the method body is known as binding.
- There are two types of binding:
- Static Binding (also known as Early Binding).
- Dynamic Binding (also known as Late Binding).



Understand the type of an instance

- 1) variables have a type
  - Each variable has a type, it may be primitive and non-primitive.

```
int data=30;
```

- 2) References have a type

```
class Dog{
 public static void main(String args[]){
  Dog d1;//Here d1 is a type of Dog
 }
}
```

- 3) Objects have a type
  - An object is an instance of particular java class, but it is also an instance of its superclass. Here d1 is an instance of Dog class, but it is also an instance of Animal.

```
class Animal{}

class Dog extends Animal{
 public static void main(String args[]){
  Dog d1=new Dog();
 }
}
```

Static binding

- When type of the object is determined at compiled time(by the compiler), it is known as static binding.
- If there is any private, final or static method in a class, there is static binding.
- Example:

```
class Dog{
 private void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Dog d1=new Dog();
  d1.eat();
 }
}
```

Dynamic binding

- When the type of the object is determined at run-time, it is known as dynamic binding.
  - The example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. Therefore the compiler does not know its type, only its base type.

```
class Animal{
 void eat(){System.out.println("animal is eating...");}
}

class Dog extends Animal{
 void eat(){System.out.println("dog is eating...");}

 public static void main(String args[]){
  Animal a=new Dog();
  a.eat();
 }
}
```

Instanceof

- The **instanceof operator** is used to test whether the object is an **instance of the specified type** (class or subclass or interface).
- It is also known as type *comparison operator* because it **compares the instance with type**. It returns either true or false. If the instanceof operator is applied with any variable that has null value, it returns false.
- An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

```
class Simple1{
 public static void main(String args
 Simple1 s=new Simple1();
 System.out.println(s instanceof Sir
 }
}
```

```
class Animal{}
class Dog1 extends Animal{//Dog inherits Animal

 public static void main(String args[]){
 Dog1 d=new Dog1();
 System.out.println(d instanceof Animal);//tru
 }
}
```

```
class Dog2{
 public static void main(String args[]){
  Dog2 d=null;
  System.out.println(d instanceof Dog2);//false
 }
}
```

Downcasting with Java instanceof operator

- When a Subclass type refers to the object of a Parent class, it is known as downcasting.
- If performed directly the compiler gives an error.

```
Dog d=new Animal();//Compilation error
```

- If performed by typecasting, ClassCastException is thrown at runtime.

```
Dog d=(Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime
```

- If instanceof operator is used, downcasting is possible (com.prg103.instanceofexamples.Dog).

Downcasting without the use of java instanceof

- Downcasting can also be performed without the use of instanceof operator (com.prg103.instanceofexamples.Dog1).
- Closer look: The actual object that is referred by a, is an object of the Dog class. If downcasted, it is fine. But what will happen if the following code is run?

```
Animal a=new Animal();
Dog.method(a);
//Now ClassCastException but not in case of instanceof operator
```

Understanding Real use of instanceof in java

- Real use of instanceof keyword by example (com.prg103.instanceofexamples.instanceofTest)
- Second example (com.prg103.instanceofexamples.instanceofTest2)
  - Short explanation of the output: Instances **car** and **moto** are also of type **Vehicle** due to hierarchy, so these assumptions are true. However, **vehicle** is not instance of **Car** (neither **MotorCycle** of course). Also, the instances **vehicle** and **moto** are not instances of **DriveCar**, as only **car** is an instance of that type. Finally, when the **car** gets the null value, it is not an instance of **Vehicle** or **Car** anymore.

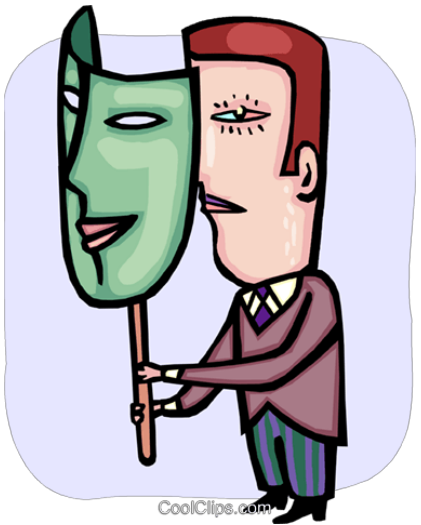

Abstract Classes,
Interfaces and
Encapsulation

Abstract classes

- A class which is declared with the **abstract keyword** is known as an abstract class.
- An abstract class can have abstract and non-abstract methods.

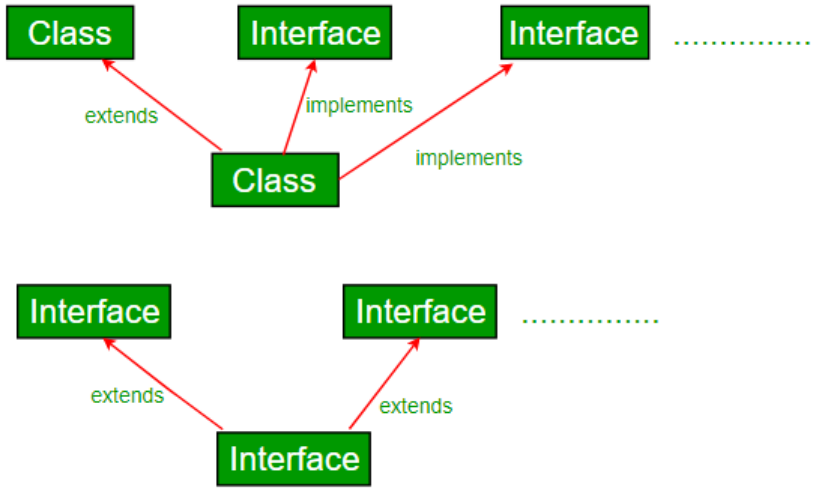
What is abstraction?

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.
- **Showing only essential parts** of your program to the user and **hide** the internal **details**.
  - For example, sending a SMS where you type the text and send the message. The user does not know the internal processes how the message is delivered.
- Abstraction leads the **focus on what the object does instead of how** it does it.



How do we achieve abstraction?

- There are basically two ways
- **Abstract classes**
  - Can model abstraction on different levels (0-100%)
- **Interfaces**
  - Are 100% abstract



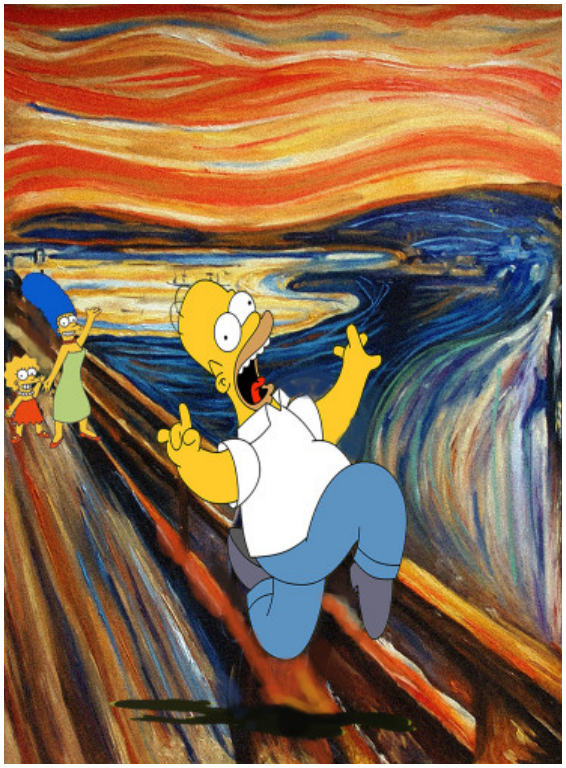
Abstract class

- As heard before, a class which is declared as abstract is known as an **abstract class**.
- Abstract classes come with some rules
  - It can consist of abstract and non-abstract methods.
  - It needs to be extended and the abstract methods need to be implemented.
  - It cannot be instantiated as an object.

```
abstract class A{}
```

Abstract classes – most important points

- An abstract class **must be declared** with an **abstract** keyword.
- It can have **abstract and non-abstract** methods.
- It **cannot** be **instantiated**.
- It **can** have **constructors** and **static methods**.
- It **can** have **final** methods
  - Which will force the subclass not to change the body of the method.



Abstract methods

- A method which is **declared** as **abstract** and does **not** have an **implementation** in the method body is known as an abstract method.

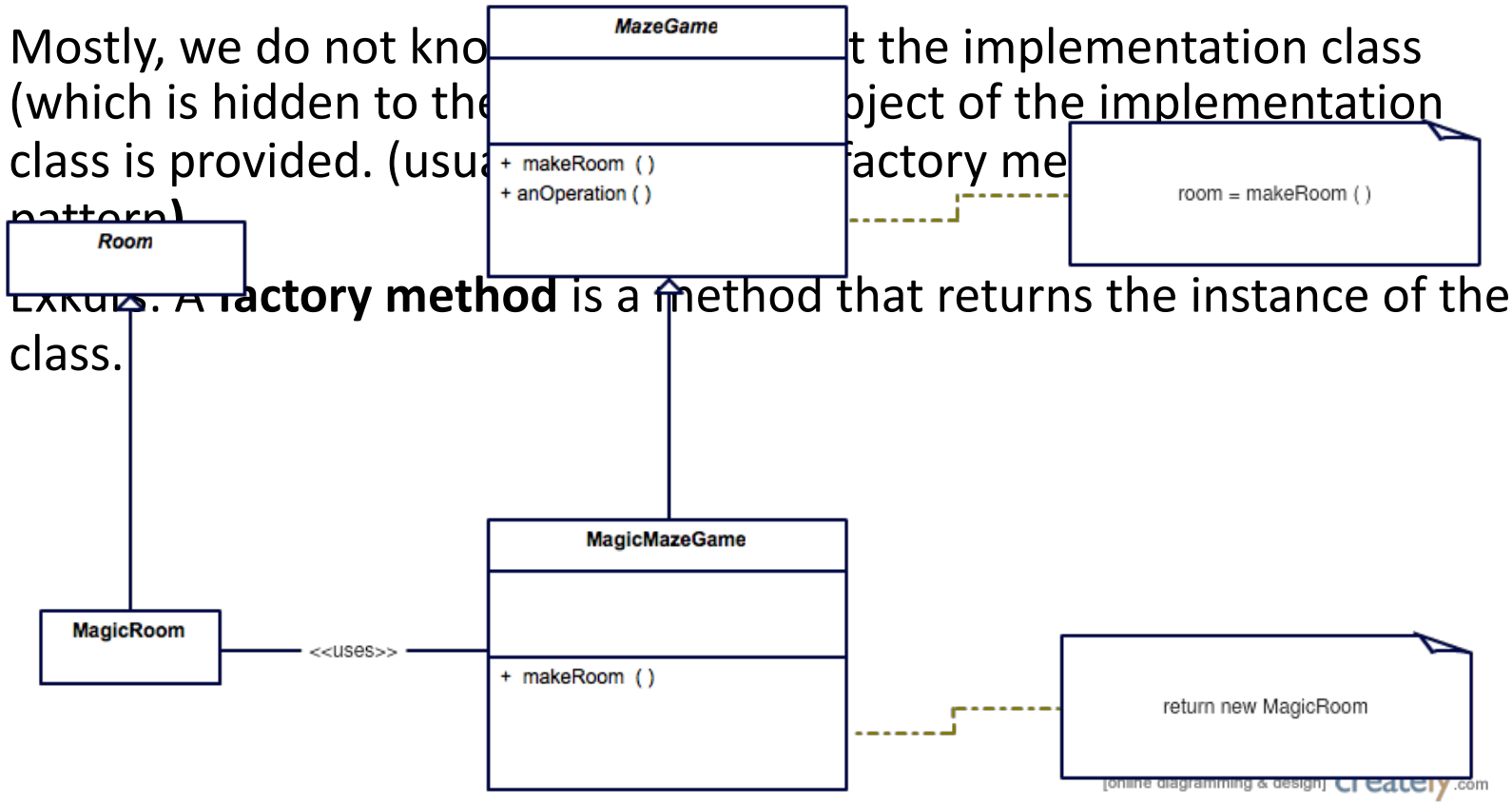
```
abstract void printStatus();//no method body and abstract
```

- Lets see an example (Bike)

Bett

"Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses." (***Design Patterns: Elements of Reusable Object-Oriented Software*** (1994))

- Shap … by the Rectangle and Circle classes.
- Mostly, we do not kno… t the implementation class (which is hidden to the … bject of the implementation class is provided. (usua… factory me… pattern)
- Exam… **factory method** is a method that returns the instance of the class.

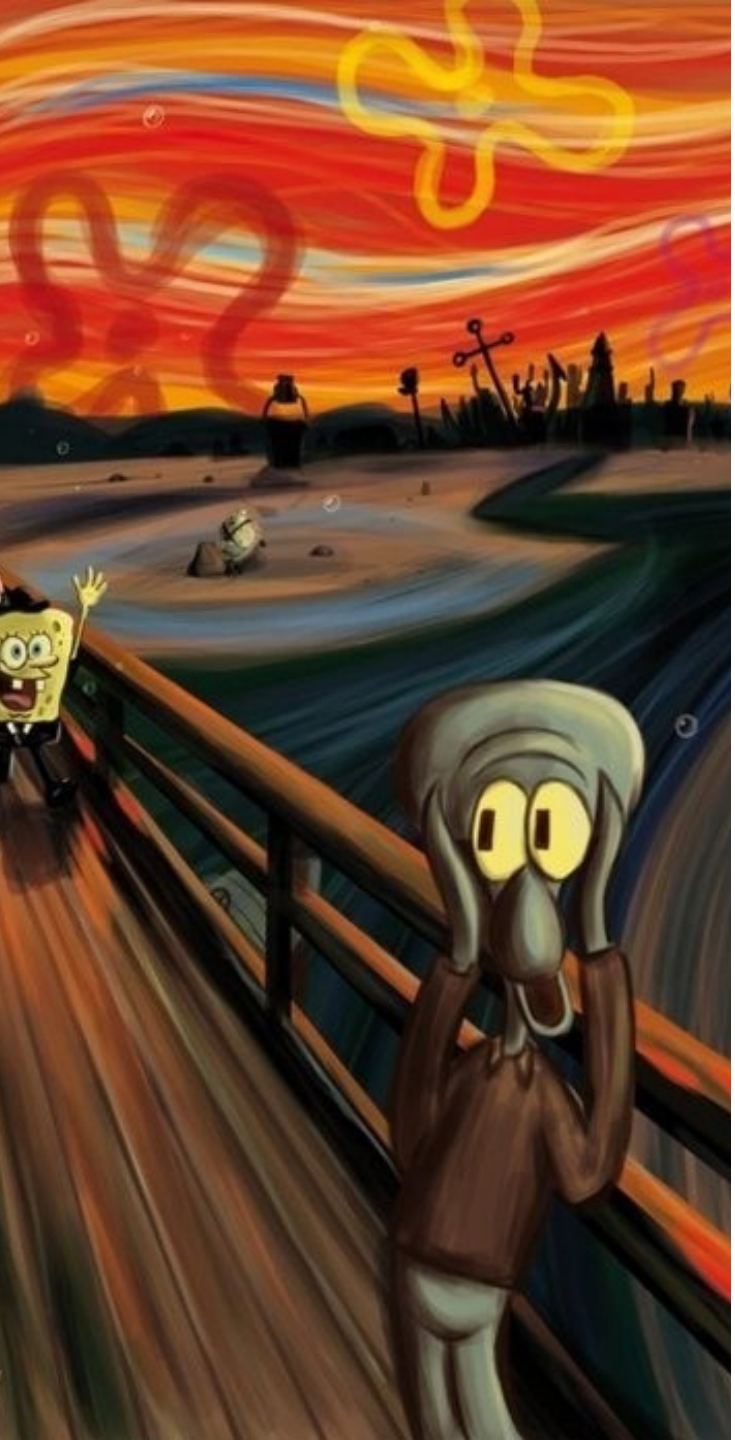


Better understanding by a real example

- Shape is the abstract class, and its implementation is provided by the Rectangle and Circle classes.
- Mostly, we do not know anything about the implementation class (which is hidden to the user), and an object of the implementation class is provided. (usually done by the factory method design pattern**)**.
- Exkurs: A **factory method** is a method that returns the instance of the class.
- In this example, if you create the instance of the Rectangle class, the draw() method of the Rectangle class will be invoked (TestAbstraction1).
- **Second example:** A banking system returning rates for different banks (TestBank).

Abstract classes - extended

- An abstract class **can have data members, abstract methods, method body** (non-abstract method), **constructors**, and even a **main()** method.
- Example: Extension of our bike example with abstract and non abstract methods (TestAbstraction2).
- **Tipp: If there is an abstract method in a class, the class must be abstract (otherwise you get compile time error).**
- **Tipp: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make the extended class abstract again.**

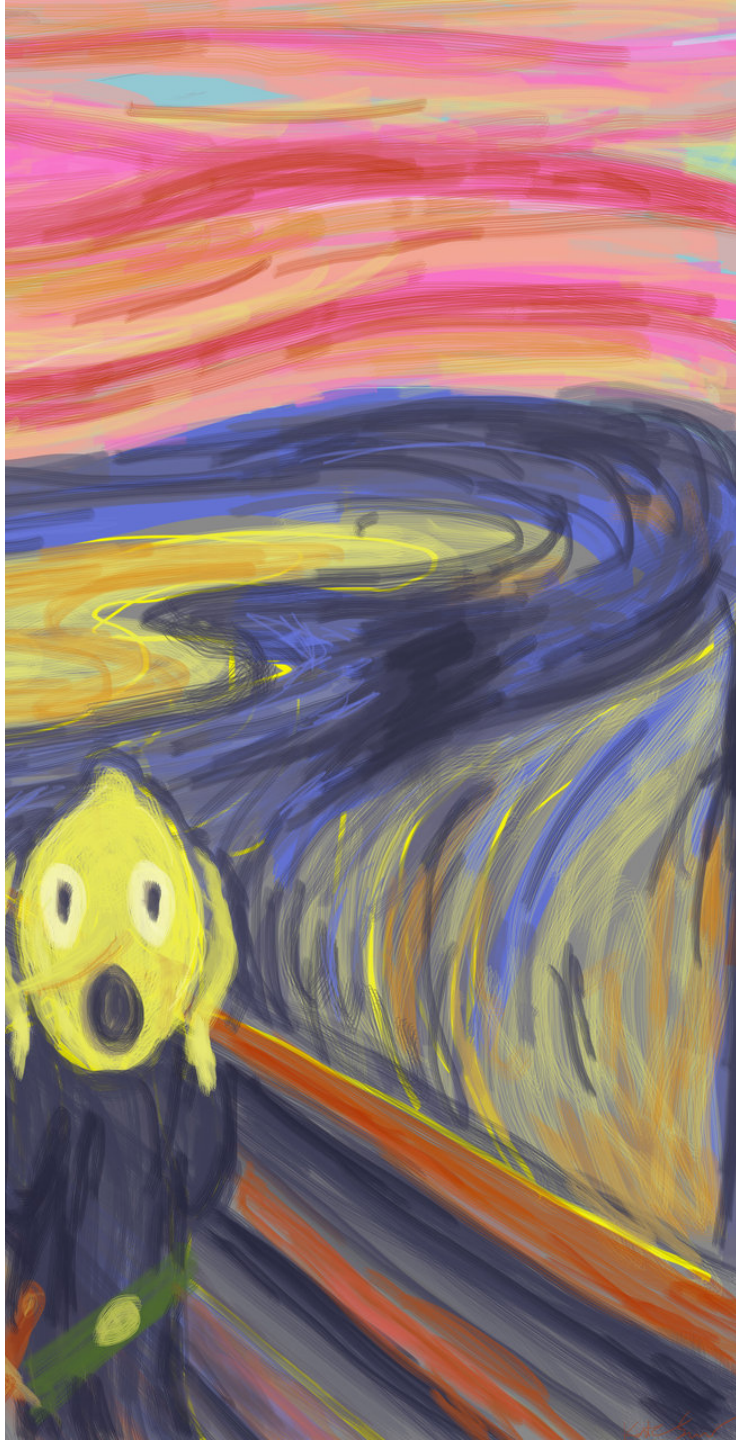


Interfaces (i)

- An **interface in java** is a blueprint of a class (blueprint of a blueprint).
  - It can have static constants and abstract methods.
- An interface is *a mechanism to achieve abstraction*.
  - Mainly only abstract methods in the interface, no method bodies (static and default methods are possible).
  - It is used to achieve abstraction and multiple inheritance in Java.
- In other words, you can say that interfaces can have abstract methods and variables but they cannot have a method body (beside of default and static).

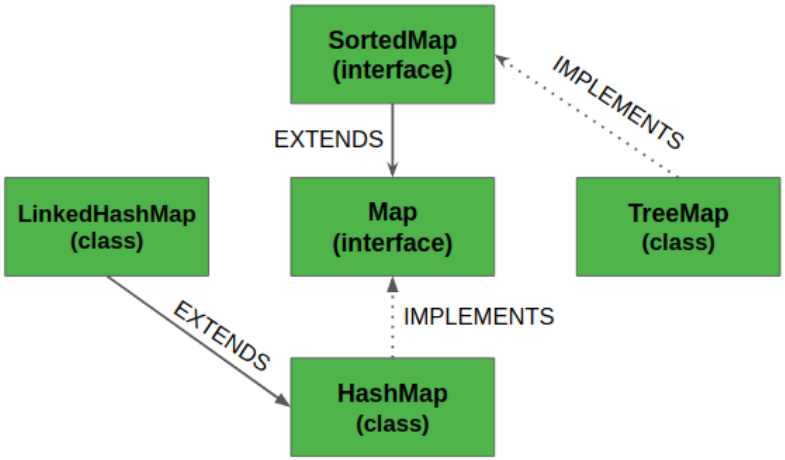
Interfaces (ii)

- Java Interface also **represents the IS-A relationship**.
- Interfaces cannot be instantiated similar to the abstract classes.
  - Since Java 8, **default and static methods can be part** of an interface.
  - Since Java 9, **private methods can be** in an interface.



Usage of interfaces

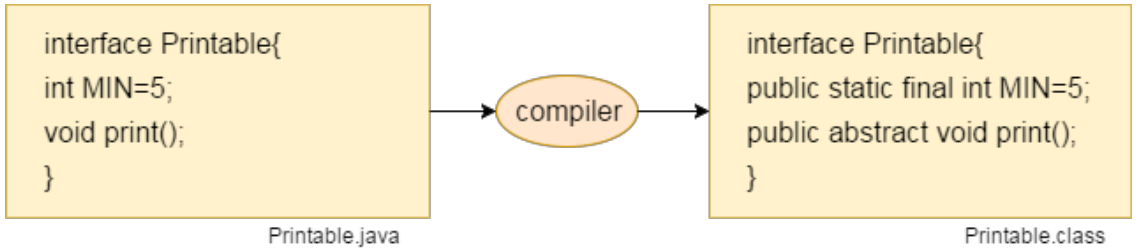
- There are mainly three reasons why to use interfaces.
- 1) To achieve **abstraction**.
- 2) Interfaces support the functionality of **multiple inheritance**.
- 3) Can be used to achieve loose coupling (classes are only connected trough interfaces, mostly independent).



MAP Hierarchy in Java

How to declare an interface

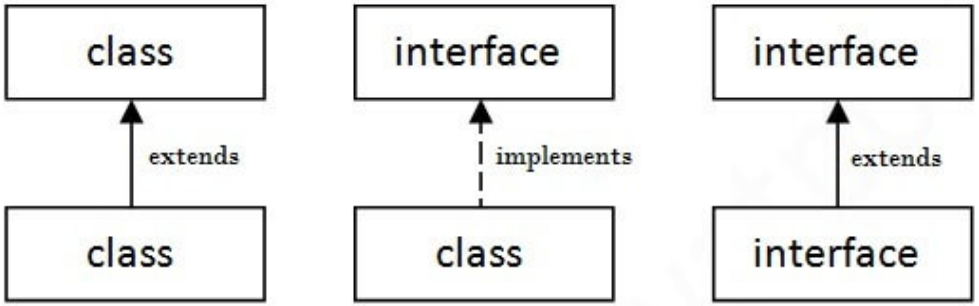
- An interface is declared by using the **interface** keyword.
- It provides **total abstraction**; meaning all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default.
- A class that implements an interface **must implement all the methods declared in the interface**.
- The Java compiler adds public and abstract keywords before the interface methods. Moreover, it adds public, static and final keywords before data members.




```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Relationship between classes and interfaces

- A class extends another class, an interface extends another interface, but a **class implements an interface**.



- Example: The Printable interface has only one method, and its implementation is provided in the Aclass class (Aclass)

Interface examples

- In this example, the **Drawable interface** has only one method. Its **implementation** is provided **by Rectangle and Circle** classes.
- In a real scenario, an interface is usually **defined by someone else**, but its implementation is provided by different implementation providers.
- Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface (TestInterface1).
- **Bank example** with interfaces (TestInterface2).

Multiple inheritance using interfaces

- If a class implements **multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance** (Aclass2).

Multiple Inheritance in Java

- Q) Multiple inheritance is not supported through class in Java, but it is possible by an interface, why?
  - As we have explained in the inheritance part, multiple inheritance is not supported in the case of a class because of **ambiguity**. However, it is supported in case of an **interface** because there is **no ambiguity**. because its implementation is provided by the implementation class (TestInterface3).

Interface inheritance

- An interface can extend another interface to achieve inheritance (Aclass2).
- **Default Method in Interface**
  - Since Java 8, we can have method bodies in interfaces, but we need to make them default methods (TestInterfaceDefault).
- **Static Method in Interface**
  - We can also have static methods in interfaces (TestInterfaceStatic).

Difference between abstract class and interface

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class**can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class**can be extended using keyword "extends".	An **interface class**can be implemented using keyword "implements".
8) A Java**abstract class**can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

- **Let's see an example (differenceTest)**

Encapsulation

- **Encapsulation** is a *process of* ***wrapping code and data together*** *into a single unit*, for example, like a capsule which is mixed of several medicines.
- We can create a **fully encapsulated** class by making **all** the data members of the class **private**.
- **Setter and getter** methods are needed to set and get the data in it.
- Why do we want to use it?
  - Encapsulation promotes **maintenance**
  - Code **changes** can be **made independently**
  - Increases **usability**

Advantage of encapsulation

- By providing only a setter or getter method, you can make the class **read-only or write-only**. In other words, you can skip the getter or setter methods to protect different part if needed.
- It provides **control over data**.
  - Suppose you want to set the value of id which should be greater than 100 only, you can write **the logic inside the setter method**. You can write the logic not to store the negative numbers in the setter methods.
- It is a way to achieve **data hiding.**
  - Other **classes will not be able to access** the data through the private data members.
- Standard IDE's are providing the facility to generate the getters and setters. Therefore, it is **easy and fast to create an encapsulated class**.

Simple example of encapsulation

- Student example with encapsulation (com.pgr103.encapsulation)
  - **Read only class** has only getter methods
  - **Write only class** has only setter methods
- Another example: Account (account and testaccount)