Class:
Blueprint/ Template

Object:
Entity/ Instance that follows the design of a class. Thus, it is of the class type that it belongs to. Ex: a student object is of student class type.

- Addresses of objects are always different from one another.

Object has 2 things:
1. Attributes (Instance Variable)
2. Behavior (Instance Method)

Both of these are written inside the design class. Objects of this class will get their own copies of these instance variables. Instance methods can be called upon the objects of the class to show their behavior.

Creating an object - in details:
BracStudent student1 = new BracStudent();

new BracStudent() -> Actually creating an object of the BracStudent class
new - allocates memory to the object created
BracStudent() - calls a constructor

BracStudent student1 --> stores the location/ reference of the object created to student1 variable
type of student1 reference variable is BracStudent because it contains the reference of a BracStudent object

Instance Variable:

- Declared inside a class just after writing the class name.
- Defines the features or attributes of an object of a class.
- Belongs to an object.
- Every object of a class will have its own copy of instance variables in the memory.
- Values of those attributes may differ from object to object.
- Instance variables can be accessed and updated inside a design class easily just by writing its variable name, ex: id or by using the this keyword when there is a local variable of the same name, ex: this.id.
- Can be accessed using object reference outside the design class. Ex: box1.height.
- Change to the instance variable of one object will not affect the instance variable of

another object.

Local Variable:

- Temporarily created inside a method when it is invoked.
- It cannot be accessed outside the method it was created in, unlike instance variables.
- The parameters of an instance method are also its local variables.

Point to remember for tracing:

- If an instance and local variable of the same name exists in a method, writing just the variable name means accessing the local variable. Ex: If there is an instance variable named sum and a local variable named sum is also created inside a method, just writing sum will refer to the local variable.
- this.variable_name will always refer to the instance variable. Ex: this.sum

Instance Method:

- A certain method that needs to be called on an object.
- Purpose is to show the behaviour of the object it is called upon.
- Can be parameterized or non parameterized.
- We access the instance variables inside an instance method, of the object that the instance method was called upon.

Ex: Test t1 = new Test();

t1.methodA();

In methodA(), if we access instance variables like this.x or x (when there is no local variable of the same name, x), it means that we are accessing the instance variable, x of t1 object.

Method Overloading

- Java allows multiple methods to have the same name as long as their parameters are different. Known as method overloading.
- Different parameters mean - **number/ type/ order** of parameters are different.
- If you have 2 methods with the same name and same parameters but their return type is different, it does not fall under the method overloading category. Java will not allow this as parameters are not different and thus will show an error.
- Even if we need to write two methods of the same name but different parameters,

sometimes, we can utilize one method from the other by calling that method. It reduces code redundancy.

Ex: Writing an addStudent() method which accepts one student name as parameter and an addStudent() method which accepts two student names. We can call the first method from the second method twice and pass the two student names,

addStudent(String name)
{
  //adds a name to an array
}
	addStudent(String name1, String name2)
	{
	  addStudent(name1);
	  addStudent(name2);
	}

However, this approach will not always work and we may need to write different code for the two methods.

Constructor

- **A special method called when an object is created/ instantiated**.
- Primary purpose - to initialize the instance variables of an object.

Constructor Overloading

- Multiple constructors of the same class with different parameters.
- We can call one constructor from another constructor by using this().

Encapsulation

- Scope of Access Modifiers

Scope		Public	Private	Protected	Default
Same Package	Same Class	Yes	Yes	Yes	Yes
	Non Subclass	Yes	No	Yes	Yes
	Sub Class	Yes	No	Yes	Yes
Different Package	Sub Class	Yes	No	Yes	No
	Non Subclass	Yes	No	No	No

- Private variables cannot be accessed outside of its own class. To read the private variable/ get its value, we need to design a get method in the class that the private variable is in, which will return the value of that private variable. Call the get method from **outside of its own class** when necessary to get the value of the private variable. To update the private variable outside of its own class, a set method is needed to be designed in the class that the private variable is in. Call the set method from **outside of its own class** when necessary to update the value of the private variable.

Difference between instance and static variable

- Static Variable - Belongs to a class.

Instance Variable	***Static Variable***
Instance variables are the attributes of an object.	Static variables are the attributes of a class.
Declared without static keyword	Must use a static keyword to declare it.
Initialized when objects are created.	Initialized when a class is loaded in memory, before even objects are created. So the existence of static variables depends on the existence of its class and not its objects.
Every object has its own copies of instance variables	There is only **a single copy of each static variable shared by all objects of a class**. That

	single copy is updated every time we update the static variable.
Any attribute that seems like the individual property of an object is to be kept as instance variables. Ex: id, name etc are individual properties of Student objects. Every Student object has these properties.	Usually, if attributes may seem like we are collecting information from all objects (ex: storing names of all objects) or counting number of objects etc, these attributes need to be kept as static variables.

- There cannot be an instance and static variable of the same name inside a class.

Static Method

- Differences between instance and static method

Static Method	***Instance Method***
Used to modify state of a class.	Used to modify state of an object.
Does not need the creation of an object to be called.	Needs the creation of an object to be called as instance methods are called on an object reference.
They can only ***directly*** access static variables of their class or update them.	Instance methods can ***directly*** access both instance and static variables.
Static methods can only ***directly*** call other static methods of their class.	They can ***directly*** call both instance and static methods of their class.
They cannot refer to ***this*** or ***super*** in any way. (*super will be covered in inheritance*)	They can refer to ***this*** or ***super*** (but not both).

How to identify Static vs Instance Method in Code?

Static Method	***Instance Method***
Called with reference to a class.	Called with reference to objects.

Ex:
 	
public class StudentTester
{
  public static void main (String [] args)
  {
  Student.print(); // print() is a static method
  Student s1 = new Student();
  s1.details(); // details() is an instance method
  }
}	

How to identify Static vs Instance Variables in Code?

Static Variable	***Instance Variable***
1. Called with reference to a class.	1. Called with reference to objects.
2. Variables from static method - static variables	2. Variables from instance method - instance variables usually but could also be static. If it seems like a shared information from all objects - static

Ex:
 	
public class StudentTester
{
  public static void main (String [] args)
  {
  System.out.println(Student.count); // count is a static variable
  Student s1 = new Student();
  s1.age = 25; // age is an instance variable
  System.out.println("1========");
  Student.print();
  System.out.println("2========");
	1========
	Student names: //static variable
	2========
	Student count: 1 //static
	Name: Default //instance
	Age: 25 //instance
	3========

s1.details();
System.out.println("3========");
}
}	

Points to remember for Local, Static and Instance Variable tracing:

- Need to create single columns of static variables of a class at the very beginning in tracing tasks.
- **class_ref.var** - static var
- **this.var** or variables called on an object reference - instance/static var
- **var** - local else instance/static var

How to Answer Inheritance Coding Questions:

Usually, parent and tester classes are given. You have to design the child class.

- Parent class objects only inherit properties of parent class (variables and methods). So for those objects, the given parent class is enough.
- Child class objects inherit all properties (variables and methods) from the parent class, however they may also need some new properties in their own class. Inheritance questions usually demand you to find the properties and work needed in the child class as parent class properties are already given.
- **extends** keywords must for child classes to inherit from the parent class.

Do's and don'ts of inheritance:

Points to Remember for Instance and Static Variables:	
1. If instance or static variables are already given in the parent class, child class objects inherit those so you don't need to create these again in the child class.	1. Find the extra instance or static variables that you need to create in the child class which you did not inherit from the parent.
2. **Private variables** of the parent class are not accessible in the child class.	2. Need to design get() method to access, set() method to update in the parent class and call those methods from the child class.
Points to Remember for Constructor:	

<table>
<tr><td>2. If a child class object is created with no arguments passed, parent class constructor with no parameters can be invoked for that object.</td><td>2. However, if more work is to be done inside the constructor and the parent class's constructor is not doing that work, we must create a constructor in the child class and do the work needed in the child class' constructor instead.

But Java enforces us to also utilize the parent class' constructor by calling it. In that case, from the child class constructor, parent class constructor must be called by writing super() or nothing if implicit calling.</td></tr>
<tr><td>3. If a child class object is created with arguments passed, the parent class constructor cannot be invoked for that object.</td><td>3. We must create a constructor in the child class to accept those arguments.But in the very first line of the child class constructor, we must call the parent class's constructor by writing super(arguments) if explicitly, super() or nothing if implicitly.</td></tr>
<tr><td colspan="2">Summary note for the above two points:
If you write a constructor in the child class, calling the parent class constructor in the very first line is mandatory.

● To explicitly call the parent class' constructor with arguments, write super(arguments).
● To implicitly call the parent class' constructor without arguments, write super() or no need to write super() at all.</td></tr>
<tr><td colspan="2">Points to Remember for Instance and Static Methods:</td></tr>
<tr><td>4. Methods provided by the parent are inherited by child class objects, so we must use them when necessary.
However, if the method provided by the parent is not doing the work we want, we can override the same method in the child class.</td><td>4. Even if we override a method in the child class, we can still utilize the parent class' method if needed by calling it using super.methodName();</td></tr>
<tr><td>Parent class Method:
details()
Name:
Age:</td><td>Child class method which overrode the parents one:
@Override
details()
super.details()
ID:</td></tr>
<tr><td colspan="2">When is super needed?
● To call the constructor of parent. Ex: super()
● Though any variable and method inherited from the parent class can be called by using super.variable or super.methodName() in the child class in any circumstance, it is most necessary when calling an overridden method/ hidden variable of the parent in the child class.
1. If we have a variable of the same name in both the parent and child class, the parent class variable becomes hidden inside the child class (Variable Hiding) and the child class variable is recognized instead. So to call the parent class variable inside the child class, we must use super.variable.</td></tr>
</table>

2. If we have a method in the parent class and we write the same method in the child class, the parent class method is now overridden (**Method Overriding**). To access the parent class' method from the child class now, we must use **super.methodName().**
   Note: Overridden method of Parent class and child class must have the same method signature.

******** We cannot use super inside the parent class. super is only used to call something of the parent from the child class. ***********

Why are variables hidden but method overridden?

- Because if variables of the same name exist in both the child and parent class, the variable of the parent class becomes hidden in the child class but it is **still accessible when we are in the scope of the parent class.**
- However, if methods of the same name exist in both the child and parent class, **the parent class' method is completely overridden for a child class object**. So even if we call the method from the parent class' scope, the child class' method will be called now..

```
public class Tester
{
  public static void main(String [] args)
  {
    B b1 = new B();    //Properties: Parent - x, mA(), mB(), Own/Child - x, mB()
    b1.mB();
    b1.mA();
  }
}
class A
{
  int x = 10;
  public void mA()
  {
    System.out.println(x);   //10…..Parent class' x variable recognized in the parent class scope
    mB();                    //Child class' mB() called as parent's one is overridden
  }
  public void mB()
  {
    System.out.println(x);
  }
}
class B extends A
{
  int x = 20;
  @Override
  public void mB()
  {
    System.out.println(x);       //20… Parents x variable hidden, so x gotten from child/own class recognized
  }
}
```

How to Answer Inheritance Tracing Questions:

- **Parent class objects - will only get properties from parent class**
- **Child class objects - will get properties from parent/ more top level classes + own/child class**

For **variables in the parent class's scope**	• In the scope of the parent class, only variables inherited from the parent are accessible. • If a variable name is written inside a method (ex: x), see if there is a **local variable of this name in that method first. If not found, it's either the instance/ static variable inherited from the parent**. • If **this.variable_name** accessed in parent class's scope, it is the **instance/ static variable inherited from the parent.** • **parentClass_reference.variable -> static variable of parent**
For **variables in the child class's scope**	• In the scope of the child class, own variables + ones inherited from the parent are accessible. • If a variable name is written inside a method (ex: x), see if there is a **local variable in that method first. If not found, look for the instance/ static variable obtained from its own class first. Else, look for the instance/ static variable inherited from the parent**. • If **this.variable_name is accessed** in the child class's scope, it is the **instance/ static variable obtained from the child class if available. If not available,** it is the **instance variable inherited from the parent** • If **super.variable_name is accessed** in the child class's scope, it is the **instance/ static variable inherited from the parent.** • **parentClass_reference.variable -> static variable of parent** • **childClass_reference.variable -> static variable of child, if not found, parents static variable**
For **method** calling, **it depends on which object's reference we are calling the method.** • For **parent class objects**, only methods from the parent class can be called. • For **child class objects, the child class method will be called if available, whether we are in the scope of the child class or the parent class. If not available, the parent class' method will be called.** • **super.methodName()** will always call the parent class' method.	

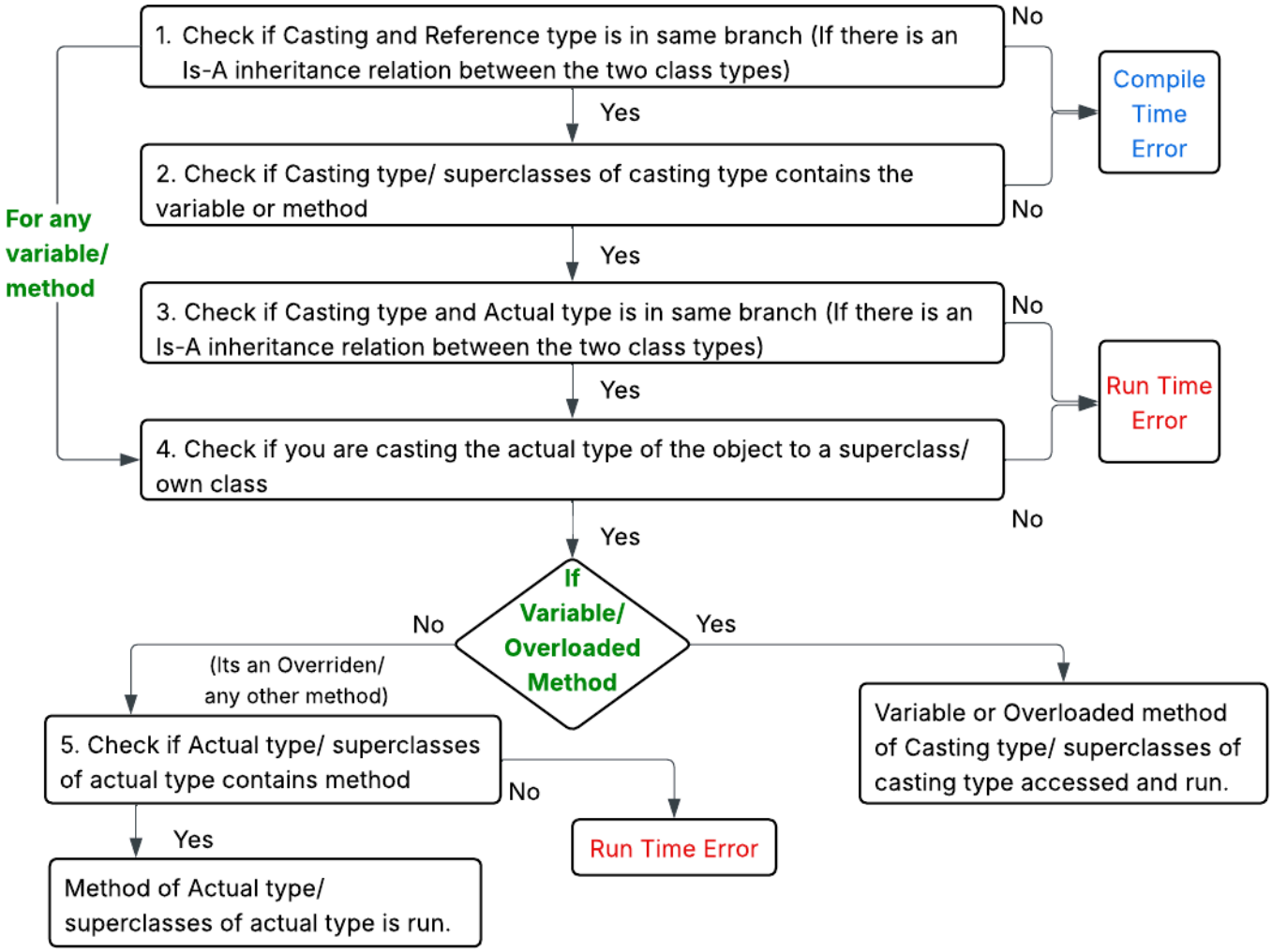
Polymorphism

Which method/ variable to call?

<table>
<tr><td colspan="2">If only Reference and Actual class type of Object present
Ex: Gadget (Reference type) m1 = new Mobile (Actual type) ();</td></tr>
<tr><td>Compile time Polymorphism Rule
(Applicable for variables, Overloaded methods)</td><td>Run time Polymorphism Rule
(Applicable for Overriden methods, Any other method not overloaded or overriden)</td></tr>
<tr><td>1. Check if Reference class type/ super classes of reference type has variable/ overloaded method. If so, code is compiled and the variable/ overloaded method of the reference type or its super classes is accessed. If not found in Reference type or its super classes, we get a Compile Time Error.</td><td>1. Check if Reference class type/ super classes of reference type has method. If so, code is compiled. If not found in Reference type or its super classes, we get a Compile Time Error.

2. Check if Actual class type/ super classes of actual type has the method. If so, the method of Actual class type/ super class is now run.</td></tr>
<tr><td></td><td></td></tr>
<tr><td colspan="2">If Reference, Casting and Actual class type of Object present

Gadget m1 = new Mobile();
((Object) m1).m2();
Here, Reference type: Gadget, Casting type: Object, Actual type: Mobile</td></tr>
</table>



When inside a design class scope:-

1. **Accessing variables -** variables inside the current class accessed/ go to super class of the existing class if not found. x/ this.x
2. **If the variable is casted, casting type's variable is now accessed.**
3. **Accessing overriden/ any other methods(not overloaded or overriden) -** Search if actual class type of the object contains method, else go to super class of actual type
4. **Accessing overloaded methods -** If casted inside the design class: Search if casting type of the object contains method, else go to super class of casting type. If not: Search method in reference type/ superclasses of reference type