Difference between inheritance and encapsulation. B5. Difference between inheritance and abstraction.

Ans :

|  |  |  |
| --- | --- | --- |
| Part | Inheritance | Encapsulation |
| Definition | A mechanism by which a new class inherits the properties and methods of an existing class. | A mechanism that binds together code and the data it manipulates. |
| Purpose | It is used for promoting code reusability  and creating a hierarchy of classes where a subclass inherits the characteristics of a superclass. | It is used to hide the internal details of  how a class or object works, exposing  only the necessary functionality while  keeping data and implementation  hidden from external access. |
| Implementation | By using the extends keyword in most programming languages. | By declaring the data members of a class as private and providing public methods to access and modify the data. |
| Benefits | \* Code reuse  \* Modularity  \* Extensibility | \* Data protection  \* Abstraction  \* Modularity |
| Access Control | Inheritance does not inherently control  access to the inherited attributes or methods. Access is determined by the  visibility of those members in the superclass. | Encapsulation provides control over  access to the internal state and methods of a class by using access modifiers like private, protected, and public. |
| Code Organization | Inheritance organizes code in a hierarchical structure, with a clear parent-child relationship. | Encapsulation organizes code within a class, grouping related data and methods together. |
| Data Protection | Inheritance doesn't inherently protect  data; it relies on the visibility modifiers  and good design practices in the superclass. | Encapsulation enforces data protection by hiding the internal state (data) and exposing only controlled interfaces (methods) for interacting with the data. |
| Complexity | Inheritance can lead to complex class  hierarchies and potential issues like the  diamond problem. | Encapsulation simplifies the complexity of classes by isolating internal implementation details and providing clear interfaces. |
| Example | A ‘Bird’ class inheriting from an ‘Animal’ class to gain common animal  attributes and methods. | An ‘Employee’ class encapsulating  details like name, salary, and methods  for interacting with employee data. Method is getter and setter. |

B5 :

|  |  |  |
| --- | --- | --- |
| Part | Inheritance | Abstraction |
| Purpose | It allows a new class to inherit  properties and behaviors from  an existing class. | It provides a way to define a blueprint or interface for a class without implementing its details. |
| Relationship | It establishes an "is-a" relationship between classes, indicating that one class is a specialized version of another. | It can be used to define common attributes and methods that multiple classes should implement, indicating a "has-a" or "can-do" relationship. |
| Usage | Often to create a hierarchy of related classes, promoting code reuse and specialization. | Used to define a common set of methods or properties that multiple classes must adhere to, ensuring consistency. |
| Implementation | The derived class inherits both  the attributes and methods of  the base class | The abstract class defines method Signatures ( without implementation), which concrete subclasses must provide. |
| Keyword | extends | abstract |
| Code Reusability | Promotes code reusability by inheriting properties and behaviors from a base class. | Promotes code reusability by defining a common interface that multiple classes can implement, ensuring a consistent structure. |
| Flexibility | Can lead to tight coupling between classes, making it harder to adapt to changes. | Provides more flexibility as it allows classes to implement the common interface in their own way, reducing coupling. |
| Body | Method in allowed body | only method or variable declaration, Body not allowed |