**Chapter # 08: The Dangers of Inheritance**

**8.1 Dangers of Inheritance:**

Inheritance in software development refers to the ability of a subclass or child class to inherit characteristics and behaviors from a parent or superclass. While inheritance can be a powerful tool in software development, allowing for code reuse and creating hierarchies of objects with common properties, it can also become dangerous when it gets out of control.

The main danger of inheritance is that it can lead to complex and tightly-coupled code that is difficult to maintain and extend over time. This is particularly true when inheritance hierarchies become deep and complex, with many layers of inheritance and multiple levels of abstraction.

Changes to the base class may have unanticipated effects on all of its child classes, which can be a problem. This may have an impact on the entire codebase, making it challenging to forecast the effects of any changes and possibly resulting in errors and other problems.

The possibility of code repetition and duplication is another risk associated with inheritance. Subclasses may wind up with more code than necessary if they inherit many attributes and behaviors from their parent classes. This could slow down the development process and make the codebase more challenging to read and manage.

Finally, inheritance can also create tight coupling between different parts of the codebase, making it more difficult to test and debug. This can be particularly problematic in large and complex applications, where bugs and other issues can be difficult to track down and fix.

**8.2 Avoid Inheritance Dangers**

To avoid these dangers, it is important to use inheritance judiciously and to keep inheritance hierarchies as simple and shallow as possible. Code should be designed with modularity and loose coupling in mind, so that changes to one part of the codebase do not have unintended consequences elsewhere. Additionally, developers should take care to avoid code duplication and to keep the codebase well-organized and easy to read and understand.

Here are some of the ways that we can use to avoid dangers of inheritance in Java.

**8.2.1 Use Composition Instead of Inheritance:**

To create complex objects from simpler ones, composition can be used in place of inheritance. According to this method, an item is constructed from other objects that can be switched in and out as necessary. Due to the fact that modifications to one item do not affect other objects in the composition, this can help prevent the Fragile Base Class problem. For instance:

|  |
| --- |
| public interface Engine {      public void start();      public void stop();  }  public class GasEngine implements Engine {      @Override      public void start() {          System.out.println("Gas engine started.");      }        @Override      public void stop() {          System.out.println("Gas engine stopped.");      }  }  public class ElectricEngine implements Engine {      @Override      public void start() {          System.out.println("Electric engine started.");      }        @Override      public void stop() {          System.out.println("Electric engine stopped.");      }  }  public class Vehicle {      private Engine engine;        public Vehicle(Engine engine) {          this.engine = engine;      }        public void startEngine() {          engine.start();      }        public void stopEngine() {          engine.stop();      }  } |

**8.2.2 Use Abstract Class and Interface:**

We can specify similar behavior for related classes using abstract classes and interfaces to save code duplication and boost flexibility. While interfaces offer a method to express a contract between classes without providing any implementation details, abstract classes offer a way to define a base implementation that can be expanded by subclasses. For instance:

|  |
| --- |
| public interface Animal {      public void eat();      public void sleep();  }  public abstract class Mammal implements Animal {      public abstract void giveSpeak();  }  public class Dog extends Mammal {      @Override      public void eat() {          System.out.println("Dog is eating.");      }        @Override      public void sleep() {          System.out.println("Dog is sleeping.");      }        @Override      public void giveSpeak() {          System.out.println("Dog is Speaking.");      }  } |

**8.2.3 Use final Keyword:**

To prohibit a class, method, or variable from being overridden or altered, use the final keyword. This can assist prevent unexpected behavior and guarantee that important system components are not altered by accident. For instance:

|  |
| --- |
| public final class ImmutableClass {      private final int value;        public ImmutableClass(int value) {          this.value = value;      }        public int getValue() {          return value;      }  }  public class MutableClass extends ImmutableClass {      public MutableClass(int value) {          super(value);      }        // This method will not compile, because ImmutableClass is final      // and cannot be extended      @Override      public int getValue() {          return super.getValue() + 1;      }  } |

**8.2.4 Follow the Liskov Substitution Principle (LSP):**

A key tenet of object-oriented programming is the Liskov Substitution Principle, which asserts that any object from a derived class should be able to be swapped out for an object from its base class without compromising the program's validity. Or, to put it another way, derived classes ought to be allowed to modify the behavior of their base classes without impairing the system's functionality. Breaking the LSP may result in bugs and ad hoc behavior. For instance:

|  |
| --- |
| public class Rectangle {      private int width;      private int height;        public Rectangle(int width, int height) {          this.width = width;          this.height = height;      }        public int getWidth() {          return width;      }        public void setWidth(int width) {          this.width = width;      }        public int getHeight() {          return height;      }        public void setHeight(int height) {          this.height = height;      }        public int getArea() {          return width \* height;      }  }  public class Square extends Rectangle {      public Square(int size) {          super(size, size);      }        @Override      public void setWidth(int width) {          super.setWidth(width);          super.setHeight(width);      }        @Override      public void setHeight(int height) {          super.setWidth(height);          super.setHeight(height);      }  }  public class Client {      public static void main(String[] args) {          Rectangle rect = new Rectangle(5, 10);          System.out.println("Rectangle area: " + rect.getArea());            Rectangle square = new Square(5);          square.setWidth(10);          System.out.println("Square area: " + square.getArea());      }  } |