**Liskov Substitution Principle**

Substitutability is a principle in object-oriented programming stating that,

In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e. an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.)

Liskov's notion of a behavioural subtype defines a notion of substitutability for objects; that is,

If S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g. correctness).

Liskov's principle imposes some standard requirements on signatures that have been adopted in newer OOP languages (usually at the level of classes rather than types):

1. Contravariance of method arguments in the subtype
2. Covariance of return types in the subtype
3. No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype

**Implementation Guidelines**

1. No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype
2. Clients should not know which specific subtype they are calling
3. New derived classes just extend without replacing the functionality of old classes

In addition to the signature requirements, the subtype must meet a number of behavioural conditions:

1. Preconditions cannot be strengthened in a subtype.
2. Postconditions cannot be weakened in a subtype.
3. Invariants of the supertype must be preserved in a subtype.
4. History constraint (the "history rule"). Objects are regarded as being modifiable only through their methods (encapsulation). Because subtypes may introduce methods that are not present in the supertype, the introduction of these methods may allow state changes in the subtype that are not permissible in the supertype. The history constraint prohibits this.

**Precondition**

A precondition is a condition or predicate that must always be true just prior to the execution of some section of code or before an operation in a formal specification.

If a precondition is violated, the effect of the section of code becomes undefined and thus may or may not carry out its intended work. Security problems can arise due to incorrect preconditions.

**Postcondition**

A postcondition is a condition or predicate that must always be true just after the execution of some section of code or after an operation in a formal specification. Postconditions are sometimes tested using assertions within the code itself. Often, postconditions are simply included in the documentation of the affected section of code.

**Invariant**

In computer science, one can encounter invariants that can be relied upon to be true during the execution of a program, or during some portion of it. It is a logical assertion that is always held to be true during a certain phase of execution. For example, a loop invariant is a condition that is true at the beginning and end of every execution of a loop.

# Example

class Employee {

int id;

string name;

public:

Employee() {};

Employee(int ID, string Name) : id(ID), name(Name) {}

double calculate\_bonus(double salary) = 0;

};

class PermanentEmployee : public Employee {

public:

PermanentEmployee() {}

PermanentEmployee(int ID, string Name) : Employee(ID, Name){}

double calculate\_bonus(double salary) {

return (salary \* 0.1);

}

};

class TemporaryEmployee : public Employee {

public:

TemporaryEmployee() {}

TemporaryEmployee(int ID, string Name) : Employee(ID, Name){}

double calculate\_bonus(double salary) {

return (salary \* 0.05);

}

};

class ContractEmployee : public Employee {

public:

ContractEmployee() {}

ContractEmployee(int ID, string Name) : Employee(ID, Name){}

double calculate\_bonus(double salary) {

throw NewException; // new exception thrown in subtype

// voilation of implementation guide

}

};

int main() {

**Employee** emp1 = new **PermanentEmployee**(1, "Emp1");

**Employee** emp2 = new **TemporaryEmployee**(2, "Emp2");

**Employee** emp3 = new **ContractEmployee**(3, "Emp3");

emp1.calculate\_bonus();

emp2.calculate\_bonus();

emp3.calculate\_bonus(); // throw exception

return 0;

}

**How to overcome this problem?**

Separate Interface for employee information and calculate bonus

class I\_Employee {

int id;

string name;

public:

I\_Employee() {};

I\_Employee(int ID, string Name) : id(ID), name(Name) {}

double get\_MinSalary() = 0;

};

class I\_EmployeeBonus {

public:

double calculate\_bonus(double salary) = 0;

};

class Employee : public I\_Employee, public I\_EmployeeBonus {

public:

Employee() {};

Employee(int ID, string Name) : I\_Employee(ID, Name) {}

double calculate\_bonus() = 0;

};

class PermanentEmployee : public Employee {

public:

PermanentEmployee() {}

PermanentEmployee(int ID, string Name) : Employee(ID, Name){}

double get\_MinSalary() { return 15000; }

double calculate\_bonus(double salary) {

return (salary \* 0.1);

}

};

class TemporaryEmployee : public Employee {

public:

TemporaryEmployee() {}

TemporaryEmployee(int ID, string Name) : Employee(ID, Name){}

double get\_MinSalary() { return 10000; }

double calculate\_bonus(double salary) {

return (salary \* 0.05);

}

};

class ContractEmployee : public I\_Employee {

public:

ContractEmployee() {}

ContractEmployee(int ID, string Name) : Employee(ID, Name){}

double get\_MinSalary() { return 5000; }

};

int main() {

Employee emp1 = new PermanentEmployee(1, "Emp1");

Employee emp2 = new TemporaryEmployee(2, "Emp2");

Employee emp3 = new ContractEmployee(3, "Emp3");

emp1.calculate\_bonus();

emp2.calculate\_bonus();

return 0;

}

# END