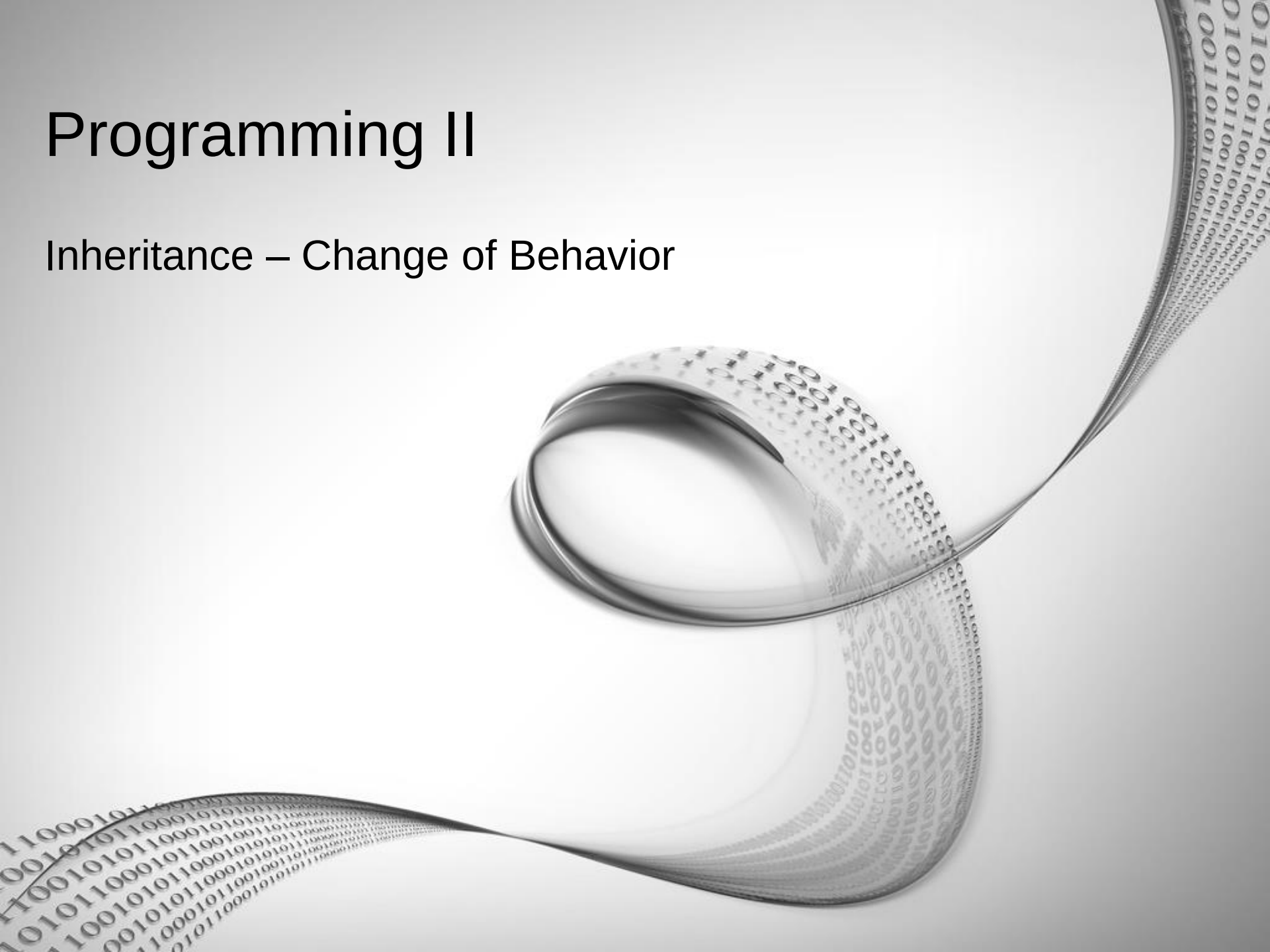
Programming II

Inheritance – Change of Behavior



Lecture Outline

- Extension of behavior
- Change of behavior
- Example

Extension of behavior

When expanding behavior...

- We can safely use what we already have.
- There is no problem with understanding how the object behaves.
- The object plays its role...
- ...of a role of one of its ancestors.

Extension-specialization paradox

- The inheritance relationship is the GENERAL-SPECIAL relationship.
- The child is therefore a special case ancestor.
- However, when the extension, it happens that the child knows more than any of its ancestor.

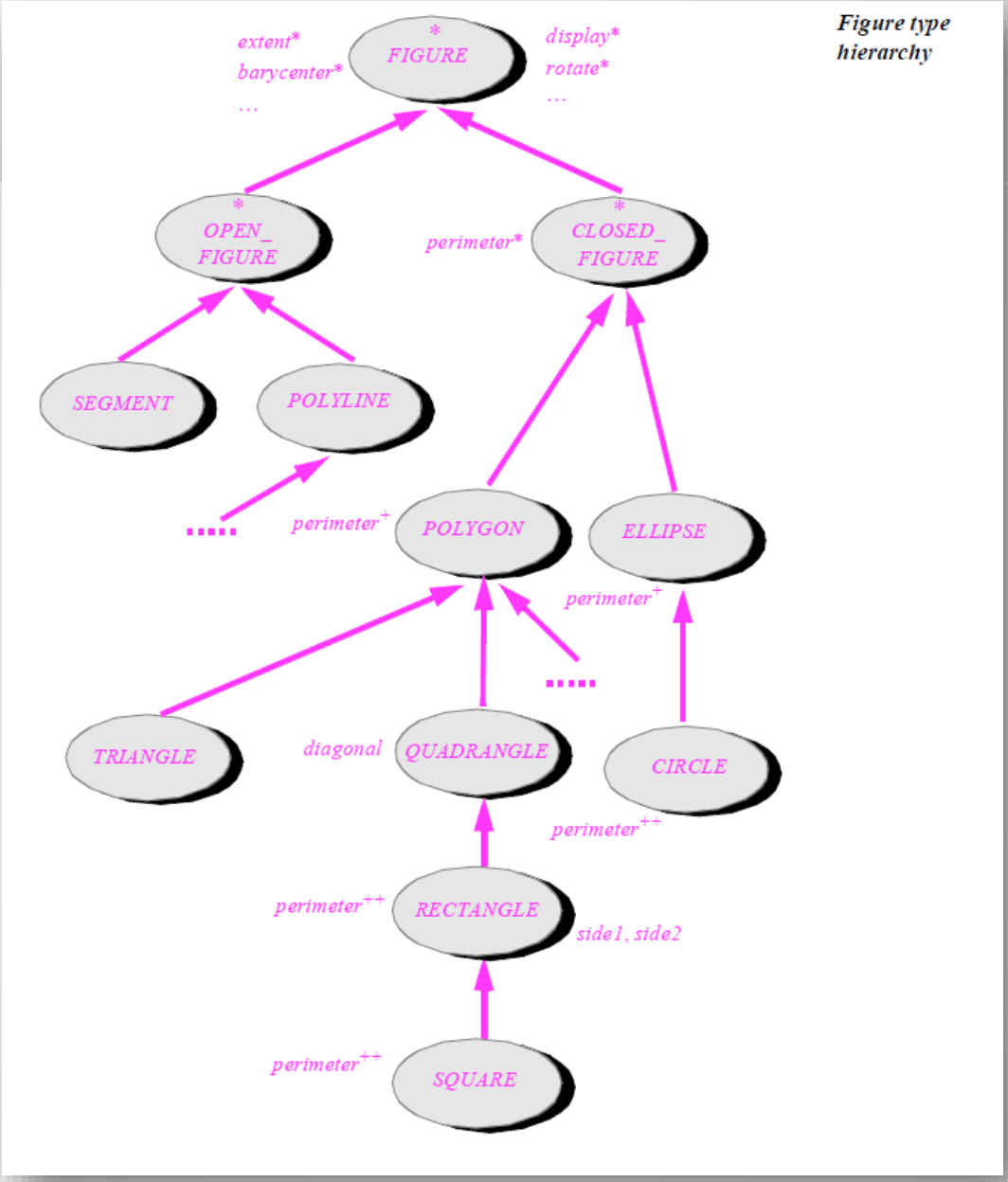


Figure type hierarchy
extent*
barycenter*
...
FIGURE
display*
rotate*
...
OPEN_ FIGURE
perimeter*
CLOSED_ FIGURE
SEGMENT
POLYLINE
perimeter+
POLYGON
ELLIPSE
perimeter+
TRIANGLE
diagonal
QUADRANGLE
CIRCLE
perimeter++
perimeter++
RECTANGLE
side1, side2
perimeter++
SQUARE

…and so:

- **The richer behavior we consider, the fewer classes it provides.**
- In the inheritance hierarchy, the least common behavior is defined in a common ancestor.
- Leaf classes of this hierarchy have the richest behavior (each slightly different).

Incorrect Example

- The need for the expansion itself is not sufficient for the use of inheritance.
- For example, in the relationship of point and circle, we need to extend the point and work with a radius of (new behavior).
- Is it correct if we decide to use inheritance?

No!!!

- The specialization is not met (the circle is not a special case of the point).

Change of behavior

Change of Behavior

- If the behavior is declared by the ancestor, we can declare the same in descendant again.
- Declared behavior should be then implemented by the descendant (to be applicable).
- Declared behavior does not have to be implemented in the ancestor.

Overloading

- By *overloading* we mean the situation when a method has the same name, but has:
  - different parameters,
  - different types of parameters,
  - different type of return value.
- Overloading is not change of behavior, even though the method has the same name.

Types of Overloading

- Method name remains the same.
- A different number of parameters.
- Different data types of parameters.
- Different return value (not in C ++).
- Everything can be combined.

Overriding

- By *overriding* we understand the situation where a child has the same method declaration as a method ancestor (the same signature).
- The descendant inherits the ancestor method. It has two methods with the same declaration.

When to Use Overriding?

- A typical application of *overloading* is constructors.
- Using *overriding* represents a real change in the behavior of a descendant.
- An example might be a method for withdrawing money from different types of bank accounts.

Example

```
class Account {
private:
    int number;
    float balance;
    float interestRate;

    Client * owner;

public:
    Account(int n, Client * o);
    Account(int n, Client * o, float ir);

    int GetNumber();
    float GetBalance();
    float GetInterestRate();
    Client * GetOwner();

    void Deposit(float c);
    bool CanWithdraw();
    float Withdraw(float c);
    void AddInterest();
};
```

Overriding

- We declare class *Credit Account*.
- We override the method *CanWithdraw*.
- It has the same signature but different definition.
- Remark: The class *CreditAccount* has the instance method *CanWithdraw* twice!!!

```
class CreditAccount : public Account{
private:
    float credit;

public:
    CreditAccount(int n, Client * o, float r);
    CreditAccount(int n, Client * o, float ir, float r);

    bool CanWithdraw(float c);
    float Withdraw(float c);
};
```

```
bool Account::CanWithdraw(float c){
    return (c <= this->balance);
}

bool CreditAccount::CanWithdraw(float c){
    return (c <= (this->GetBalance() + this->credit));
}
```

Is It Done?

- NO!!!
- How to withdraw (if we can) when we do not have access to the member variable *balance*?
- What are our possibilities?

```
float Account::Withdraw(float c){
    if (c <= this->balance){
        this->balance -= c;
        return c;
    }
    return 0;
}
```

So what are our options?

- Public access to the data field?
- Violation of encapsulation?
- Some alternatives?

```
class Account {
private:
    int number;
    float balance;
    float interestRate;

    Client * owner;

public:
    Account(int n, Client * o);
    Account(int n, Client * o, float ir);

    int GetNumber();
    float GetBalance();
    void SetBalance(float c);
    float GetInterestRate();
    Client * GetOwner();

    void Deposit(float c);
    bool CanWithdraw(float c);
    float Withdraw(float c);
    void AddInterest();
};
```

```
class Account {
private:
    int number;
    float interestRate;
    Client * owner;

protected:
    float balance;

public:
    Account(int n, Client * o);
    Account(int n, Client * o, float ir);

    int GetNumber();
    float GetBalance();
    float GetInterestRate();
    Client * GetOwner();

    void Deposit(float c);
    bool CanWithdraw(float c);
    float Withdraw(float c);
    void AddInterest();
};
```

```
class CreditAccount : public Account{
private:
    float credit;

public:
    CreditAccount(int n, Client * o, float r);
    CreditAccount(int n, Client * o, float ir, float r);

    bool CanWithdraw(float c);
    float Withdraw(float c);
};
```

```
float CreditAccount::Withdraw(float c){
    if (c <= (this->GetBalance() + this->credit)){
        this->balance -= c;
        return c;
    }
    return 0;
}
```

Violation of Encapsulation

- When we change the behavior, we may need to work with the private members of the ancestor.
- This is obviously a violation of encapsulation and we must be careful...
- ... However, any reasonable rule can have some exceptions.

…otherwise?

```
float Account::Withdraw(float c){
    if (this->CanWithdraw(c)){
        this->balance -= c;
        return c;
    }
    return 0;
}
```

Can we call method of ancestor?

- It is the same as calling a static method ☹
- We call the ancestor method from its descendant.
- *Account::CanWithdraw(c);*

Sources

- Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall 1997. [459-467]

Questions

- What do we understand by the extension-specialization paradox?
- Provide examples of correct and incorrect relationship "generalization-specialization."
- What do we understand by the change of behavior in inheritance?
- What do we mean overloading? Is it an extension or change of behavior?
- Provide various types of overloading.
- What do we mean overriding? Is it an extension or change of behavior?
- What principle is violated by using "protected" keywords and why?
- What problems does the need for a change of behavior bring in inheritance?
- Regarding inheritance, describe problems of different level of access to members of the class.
- How using "protected" influence the relation between an ancestor and its descendants?