CSE 1020: Unit 9

Topics: Inheritance

To do: Chapter 9; Lab 9

1

Outline

- **Motivation**
- **Subclass API**
- **Substitutability and Polymorphism**
- **Abstract classes and interfaces**
- **The class Object**

2

Outline

- **Motivation**
- **Subclass API**
- **Substitutability and Polymorphism**
- **Abstract classes and interfaces**
- **The class Object**

3

Motivation: What and why

What

- Occasion often arises where we need to define a class that is similar to an existing one.
  - Perhaps we just need to add a few new attributes.
  - Perhaps we just need to add a few new methods.

4

Motivation: What and why

What

- Occasion often arises where we need to define a class that is similar to an existing one.
  - Perhaps we just need to add a few new attributes.
  - Perhaps we just need to add a few new methods.
- OOP supports the definition of a new class as a subclass (or specialization) of an old one.
  - Rather than requiring that an entirely new class be defined from scratch.

5

Motivation: What and why

What

- Occasion often arises where we need to define a class that is similar to an existing one.
  - Perhaps we just need to add a few new attributes.
  - Perhaps we just need to add a few new methods.
- OOP supports the definition of a new class as a subclass (or specialization) of an old one.
  - Rather than requiring that an entirely new class be defined from scratch.
- The subclass inherits all of the attributes and methods of the old class
  - It can add new attributes and methods.
  - It can even override old methods.

6

Motivation: What and why

What

- Occasion often arises where we need to define a class that is similar to an existing one.
  - Perhaps we just need to add a few new attributes.
  - Perhaps we just need to add a few new methods.
- OOP supports the definition of a new class as a subclass (or specialization) of an old one.
  - Rather than requiring that an entirely new class be defined from scratch.
- The subclass inherits all of the attributes and methods of the old class
  - It can add new attributes and methods.
  - It can even override old methods.
- We refer to this as inheritance.

7

Motivation: What and why

Why

- Code reuse is important in software engineering.
  - This can save much time and effort during initial design and implementation.
  - It also helps in keeping all code “in sync” as subsequent modifications are required.

8

Motivation: What and why

Why

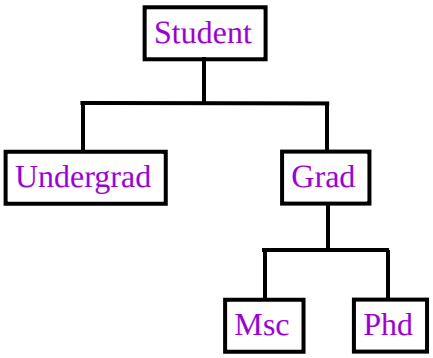
- Code reuse is important in software engineering.
  - This can save much time and effort during initial design and implementation.
  - It also helps in keeping all code “in sync” as subsequent modifications are required.
- Inheritance supports abstraction from general to more specific (specialized) data types.
  - A natural extension to class abstraction.
  - Yields type consolidation.

9

Example: Student

A hierarchy of classes

- The class Undergrad is a subclass of Student.
- The class Student is a superclass of Undergrad.

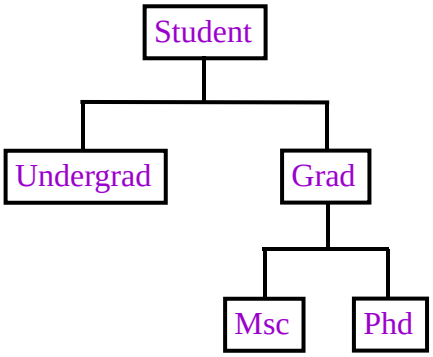


10

Example: Student

A hierarchy of classes

- The class Undergrad is a subclass of Student.
- The class Student is a superclass of Undergrad.
- Similarly
  - Grad is a subclass of Student
  - Student is a superclass of Grad

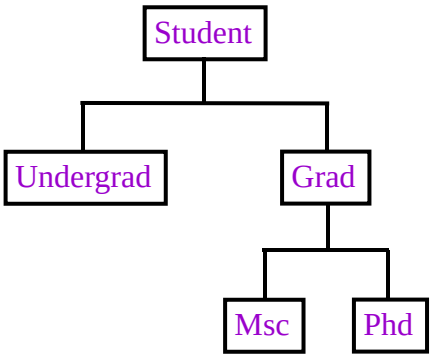


11

Example: Student

A hierarchy of classes

- The class Undergrad is a subclass of Student.
- The class Student is a superclass of Undergrad.
- Similarly
  - Grad is a subclass of Student
  - Student is a superclass of Grad
- Further
  - Msc is a subclass of Grad
  - Grad is a superclass of Msc
- Etcetera

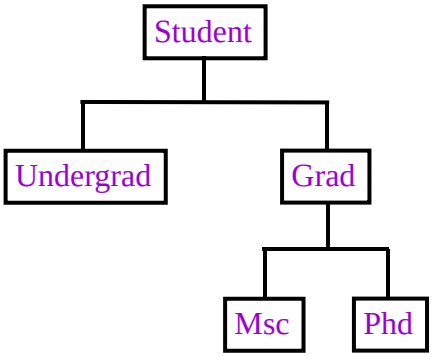


12

Example: Student

Instance of relationships

- Every instance of a class also is an instance of all its superclasses.
- An instance of Undergrad also is an instance of Student.
- An instance of Msc is an instance of Grad and an instance of Student.

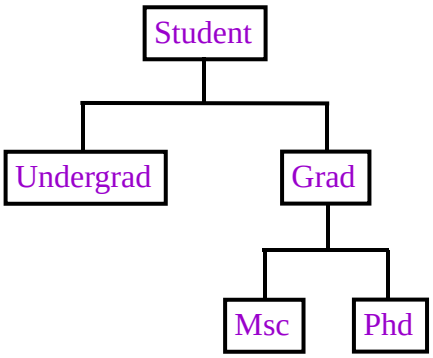


13

Example: Student

Instance of relationships

- Every instance of a class also is an instance of all its superclasses.
- An instance of Undergrad also is an instance of Student.
- An instance of Msc is an instance of Grad and an instance of Student.
- We see that the relationships are similar to those of sets.

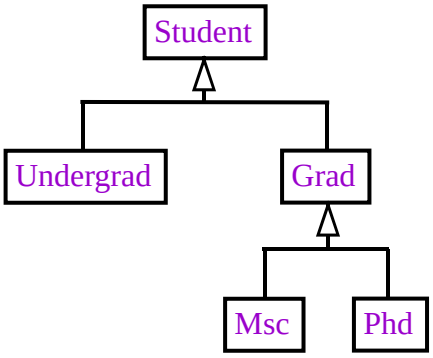


14

Example: Student

Instance of relationships

- Every instance of a class also is an instance of all its superclasses.
- An instance of Undergrad also is an instance of Student.
- An instance of Msc is an instance of Grad and an instance of Student.
- We see that the relationships are similar to those of sets.
- Some speak of an "is-a" relationship.



UML representation

15

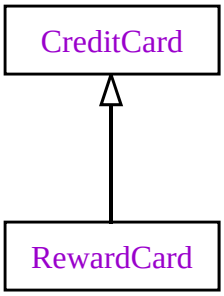
Example: CreditCard/RewardCard

Instance of relationships

- Class RewardCard is a subclass of CreditCard.
- It inherits CreditCard's methods
  - e.g., getNumber, pay, etc.

  and fields/attributes.
- Class RewardCard also defines some new methods
  - e.g., getRewardPoints

  and new fields/attributes.



UML representation

16

Outline

- Motivation
- **Subclass API**
- Substitutability and Polymorphism
- Abstract classes and interfaces
- The class Object

17

Subclass API

Header

- The API header explicitly notes the heritage of a class through use of the Java keyword extends

```
public class RewardCard
extends CreditCard
```

18

Subclass API

Header

- The API header explicitly notes the heritage of a class through use of the Java keyword extends

```
public class RewardCard
extends CreditCard
```

- Notice that all class APIs we have seen use extends through the presence of

```
extends Object
```

- We shall see that in Java, all classes are descends (subclasses of) the (super)class Object.

19

Subclass API

Constructor

- As a user, we see nothing new in the API constructor section.
- In Java, constructors are *not* inherited.

20

Subclass API

Constructor

- As a user, we see nothing new in the API constructor section.
- In Java, constructors are *not* inherited.

Constructor Summary

RewardCard(int no, java.lang.String aName)

Construct a reward card having the passed number and holder name, and set its initial dollar and point balances to zero.

RewardCard(int no, java.lang.String aName, double aLimit)

Construct a reward card having the passed number, holder name and credit limit and set its initial dollar and point balances to zero.

21

Subclass API

Methods

- By default, a subclass, inherits the methods of the superclass.
- The inherited methods are noted beneath the method summary table.

22

Subclass API

Methods

- By default, a subclass, inherits the methods of the superclass.
- The inherited methods are noted beneath the method summary table.

Method Summary

…

Methods inherited from class type.lang.CreditCard

getBalance, getExpiryDate, getIssueDate, getLimit, getName, getNumber, isSimilar, pay, setExpiryDate, setLimit

23

Subclass API

Methods

- Given that we have bothered to consider a subclass, which implies some specialization of the superclass,…
- …it is only natural to find that the methods of the superclass do not completely satisfy the requirements of the subclass.
- There are three situations to distinguish
  1. New methods
  2. Cross-class overloading of methods
  3. Overriding methods

24

Subclass API

Methods: New methods

- Problem: In certain situations, the behaviour of the superclass simply does not encompass that of the subclass.
  - For example, CreditCard has no notion of reward points, which are definitive of RewardCard.

25

Subclass API

Methods: New methods

- Problem: In certain situations, the behaviour of the superclass simply does not encompass that of the subclass.
  - For example, CreditCard has no notion of reward points, which are definitive of RewardCard.
- Solution:
  - New methods are provided for the subclass.
  - Document in the subclass Method Summary.

26

Subclass API

Methods: New methods

- Example

Method Summary

int getPointBalance()

Return the number of reward points accumulated on this reward card.

void redeem(int point)

Redeem the passed number of points and reduce the point balance accordingly.

27

Subclass API

Methods: Cross-class overloading of methods

- Problem:
  - The superclass offers methods that come close to modeling the behaviour of the subclass, …
  - …but the methods offered do not allow us to pass appropriate arguments.
- For example,
  - Class CreditCard offers an isSimilar method to compare CreditCards with parameter type CreditCard; …
  - …whereas, we want to pass an argument of type RewardCard to compare RewardCards.

28

Subclass API

Methods: Cross-class overloading of methods

- Solution:
  - Overload the method in the subclass: Provide a method of the same name, but with different signature.
  - Both methods will be documented in the API of the subclass (as standard with overloaded methods): The new one in the Method Summary; the original under Methods Inherited From.

29

Subclass API

Methods: Cross-class overloading of methods

- Solution:
  - Overload the method in the subclass: Provide a method of the same name, but with different signature.
  - Both methods will be documented in the API of the subclass (as standard with overloaded methods): The new one in the Method Summary; the original under Methods Inherited From.
- Remark: Since the two methods accomplish similar tasks (e.g., comparison of cards), it is desirable to keep the method name the same.

30

Subclass API

Methods: Cross-class overloading methods

- Example

Method Summary

boolean isSimilar(RewardCard other)
Test the similarity of two reward cards

Methods inherited from class type.lib.CreditCard

… isSimilar …

31

Subclass API

Methods: Overriding methods

- Problem:
  - The superclass offers methods that come close to modeling the behaviour of the subclass, …
  - …that allow us to pass appropriate arguments, …
  - …but somehow fall short of the needs of the subclass.
- For example,
  - Class CreditCard offers a charge method…
  - but the method does not allow us to adjust reward points, is integral to the operation of a RewardCard.

32

Subclass API

Methods: Overriding methods

- Solution:
  - Override the method in the subclass: Provide a method of the same signature and return, but the internal operations altered to model that of the subclass.
  - Only the overriding method will be documented in the API of the subclass.

33

Subclass API

Methods: Overriding methods

- Solution:
  - Override the method in the subclass: Provide a method of the same signature and return, but the internal operations altered to model that of the subclass.
  - Only the overriding method will be documented in the API of the subclass.
- Remark: Since the two methods accomplish similar tasks (e.g., charge to a card), it is desirable to keep the method name the same.

34

Subclass API

Methods: Overriding methods

- Example

Method Summary

boolean charge(double amount)
...

void credit(double amount)
…

boolean equals(java.lang.Object other)
…

String toString()
…

35

Subclass API

Overridden vs. overloaded methods and constructors

- It is important to distinguish between overloaded and overridden methods.
- Let's review the differences.

36

Subclass API

We often overload constructors

- Example

Constructor Summary

RewardCard(int no, java.lang.String aName)
Construct a reward card having the passed number and holder name, and set its initial dollar and point balances to zero.

RewardCard(int no, java.lang.String aName, double aLimit)
Construct a reward card having the passed number, holder name and credit limit and set its initial dollar and point balances to zero.

37

Subclass API

We also overload methods

- Example

Method Summary

boolean isSimilar(RewardCard other)
Test the similarity of two reward cards

Methods inherited from class type.lib.CreditCard

… isSimilar …

38

Subclass API

Overloaded methods & constructors *must* have distinct signatures

- When overloading methods we define several methods with the same name that are available in the same class.
- This is only possible when the signatures of the methods are different.
- To decide which overloaded method to call, the compiler looks at the number and type of arguments.
- If the signatures were the same, it could not determine which method to call.

39

Subclass API

Overridden methods have the same signature

- The method in the subclass replaces that in the superclass.

40

Subclass API

Example methods **Example use**

Subclass API

Example methods **Example use**

class Parent with methods
void meth( ) // #1

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

Example use

43

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

44

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
```

45

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
    Offspring( );
```

46

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
    Offspring( );
o1.meth( );
```

47

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
    Offspring( );
o1.meth( );    // calls #1
```

48

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
    Offspring( );
o1.meth( );    // calls #1
o1.meth(31);
```

49

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
    Offspring( );
o1.meth( );    // calls #1
o1.meth(31); // calls #2
```

50

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
    Offspring( );
o1.meth( );    // calls #1
o1.meth(31); // calls #2
o2.meth( );
```

51

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
    Offspring( );
o1.meth( );    // calls #1
o1.meth(31); // calls #2
o2.meth( );    // calls #1
```

52

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
    Offspring( );
o1.meth( );    // calls #1
o1.meth(31); // calls #2
o2.meth( );    // calls #1
o2.meth(29);
```

53

Subclass API

Example methods

class Parent with methods
void meth( ) // #1
void meth(int n) // #2, overloads #1

class Offspring extends Parent with methods
void meth(int n) // #3, overrides #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
    Offspring( );
o1.meth( );    // calls #1
o1.meth(31); // calls #2
o2.meth( );    // calls #1
o2.meth(29); // calls #3
```

54

Subclass API

Overridden vs. overloaded methods and constructors (recap.)

- It is important to distinguish between overloaded and overridden methods.

Subclass API

Overridden vs. overloaded methods and constructors (recap.)

- It is important to distinguish between overloaded and overridden methods.
- **Overloading:** We declare several versions of a method that work with different types of arguments.
  - Multiple versions of the method are simultaneously visible.

Subclass API

Overridden vs. overloaded methods and constructors (recap.)

- It is important to distinguish between overloaded and overridden methods.
- **Overloading:** We declare several versions of a method that work with different types of arguments.
  - Multiple versions of the method are simultaneously visible.
- **Overriding:** We declare methods that have the same signature, with a superclass/subclass relationship.
  - The overriding method replaces the overridden method.

57

Subclass API

Method API (recap.)

- We have see
  - Inheritance of methods from the superclass
  - Definition of new methods by the subclass
  - Cross-class overloading of methods
  - Overriding of methods

58

Subclass API

Attributes

- By default, a subclass, inherits the attributes (fields) of the superclass.
- The inherited attributes are noted beneath the field summary table.

59

Subclass API

Attributes

- By default, a subclass, inherits the attributes (fields) of the superclass.
- The inherited attributes are noted beneath the field summary table.

Field Summary

...

Fields inherited from class type.lang.CreditCard

MIN_NAME_LENGTH, MOD, SEQUENCE_NUMBER_LENGTH

60

Subclass API

Attributes

- Given that we have bothered to consider a subclass, which implies some specialization of the superclass,…
- …it is only natural to find that the attributes of the superclass do not completely satisfy the requirements of the subclass.
- There are two situations to distinguish
  1. New attributes
  2. Shadowing of attributes

61

Subclass API

Attributes: New attributes

- Problem: In certain situations, the data represented by the superclass simply does not encompass that of the subclass.
  - For example, CreditCard has no notion of reward points, which are definitive of RewardCard.

62

Subclass API

Attributes: New attributes

- Problem: In certain situations, the data represented by the superclass simply does not encompass that of the subclass.
  - For example, CreditCard has no notion of reward points, which are definitive of RewardCard.
- Solution:
  - New attributes are provided for the subclass.
  - Document in the subclass Field Summary.

63

Subclass API

Attributes: New attributes

- Example

Field Summary

static int REWARD_RATE

The rate used to compute the number of reward points.

64

Subclass API

Attributes: Shadowed attributes

- Problem: In certain situations, the data represented by the superclass somehow misrepresents what is intended for the subclass.
  - For example, CreditCard and RewardCard are defined to have different default credit limits.

Subclass API

Attributes: Shadowed attributes

- Problem: In certain situations, the data represented by the superclass somehow misrepresents what is intended for the subclass.
  - For example, CreditCard and RewardCard are defined to have different default credit limits.
- Solution:
  - Provide an attribute in the subclass with the same name as that of the superclass. We say that the attribute of the subclass shadows that of the superclass and that the superclass attribute is shadowed by the subclass attribute.
  - The attribute defined by the subclass will be accessed within the subclass.
  - Document in the subclass Field Summary. Do *not* list the shadowed attribute in Fields Inherited From table.

Subclass API

Attributes: Shadowed attributes

- Example

Field Summary

static double DEFAULT_LIMIT

The default credit limit used by the two-argument constructor.

67

Subclass API

Example attributes

Example use

68

Subclass API

Example attributes **Example use**

class Parent with attributes
type att1 // #1
type att2 // #2

69

Subclass API

Example attributes **Example use**

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes

70

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

71

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
```

72

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
        Offspring( );
```

73

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
        Offspring( );
o1.att1;
```

74

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
        Offspring( );
o1.att1;    // accesses #1
```

75

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
        Offspring( );
o1.att1;    // accesses #1
o1.att2;
```

76

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
        Offspring( );
o1.att1;    // accesses #1
o1.att2;    // accesses #2
```

77

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
        Offspring( );
o1.att1;    // accesses #1
o1.att2;    // accesses #2
o2.att1;
```

78

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
        Offspring( );
o1.att1;    // accesses #1
o1.att2;    // accesses #2
o2.att1;    // accesses #1
```

79

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
        Offspring( );
o1.att1;    // accesses #1
o1.att2;    // accesses #2
o2.att1;    // accesses #1
o2.att2;
```

80

Subclass API

Example attributes

class Parent with attributes
type att1 // #1
type att2 // #2

class Offspring extends Parent with attributes
type att2 // #3, shadows #2

Example use

```
// in app can write
Parent o1 = new Parent();
Offspring o2 = new
        Offspring( );
o1.att1;    // accesses #1
o1.att2;    // accesses #2
o2.att1;    // accesses #1
o2.att2;    // accesses #3
```

81

Subclass API

Attribute (field) API (recap.)

- We have see
  - Inheritance of attributes from the superclass
  - Definition of new attributes by the subclass
  - Shadowing of superclass attributes by the subclass

82

Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {
```


Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {  CreditCard cc1 = new CreditCard(703, “John”);
```

Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {  CreditCard cc1 = new CreditCard(703, “John”);
     output.println(cc1.toString());
     output.println(“credit limit: “ + cc1.DEFAULT_LIMIT);
     output.println(“name: ” + cc1.getName());
     output.println(“bal: ” + cc1.getBalance());
```

85

Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {  CreditCard cc1 = new CreditCard(703, “John”);
     output.println(cc1.toString());
     output.println(“credit limit: “ + cc1.DEFAULT_LIMIT);
     output.println(“name: ” + cc1.getName());
     output.println(“bal: ” + cc1.getBalance());
     cc1.charge(120.0);
```

86

Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {  CreditCard cc1 = new CreditCard(703, “John”);
     output.println(cc1.toString());
     output.println(“credit limit: “ + cc1.DEFAULT_LIMIT);
     output.println(“name: ” + cc1.getName());
     output.println(“bal: ” + cc1.getBalance());
     cc1.charge(120.0);
     cc1.charge(70.0);
```

87

Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {  CreditCard cc1 = new CreditCard(703, “John”);
     output.println(cc1.toString());
     output.println(“credit limit: “ + cc1.DEFAULT_LIMIT);
     output.println(“name: ” + cc1.getName());
     output.println(“bal: ” + cc1.getBalance());
     cc1.charge(120.0);
     cc1.charge(70.0);
     output.println(“bal: “ + cc1.getBalance());
```

88

Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {  CreditCard cc1 = new CreditCard(703, “John”);
     output.println(cc1.toString());
     output.println(“credit limit: “ + cc1.DEFAULT_LIMIT);
     output.println(“name: ” + cc1.getName());
     output.println(“bal: ” + cc1.getBalance());
     cc1.charge(120.0);
     cc1.charge(70.0);
     output.println(“bal: “ + cc1.getBalance());
     cc1.pay(50.0);
```

89

Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {  CreditCard cc1 = new CreditCard(703, “John”);
     output.println(cc1.toString());
     output.println(“credit limit: “ + cc1.DEFAULT_LIMIT);
     output.println(“name: ” + cc1.getName());
     output.println(“bal: ” + cc1.getBalance());
     cc1.charge(120.0);
     cc1.charge(70.0);
     output.println(“bal: “ + cc1.getBalance());
     cc1.pay(50.0);
     output.println(“bal: ” + cc1.getBalance());
```

90

Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {  CreditCard cc1 = new CreditCard(703, “John”);
     output.println(cc1.toString());
     output.println(“credit limit: “ + cc1.DEFAULT_LIMIT);
     output.println(“name: ” + cc1.getName());
     output.println(“bal: ” + cc1.getBalance());
     cc1.charge(120.0);
     cc1.charge(70.0);
     output.println(“bal: “ + cc1.getBalance());
     cc1.pay(50.0);
     output.println(“bal: ” + cc1.getBalance());
     // all of the above accesses the class CreditCard
```

91

Subclass API

Example usage

```
// assume the usual
import type.lib.*;
public class CardTest
{ public static void main(String[ ] args)
  {  CreditCard cc1 = new CreditCard(703, “John”);
     output.println(cc1.toString());
     output.println(“credit limit: “ + cc1.DEFAULT_LIMIT);
     output.println(“name: ” + cc1.getName());
     output.println(“bal: ” + cc1.getBalance());
     cc1.charge(120.0);
     cc1.charge(70.0);
     output.println(“bal: “ + cc1.getBalance());
     cc1.pay(50.0);
     output.println(“bal: ” + cc1.getBalance());
     // all of the above accesses the class CreditCard
     // continued on next page
```

92

Subclass API

Example usage

```
    // continued from previous slide




  }
}
```


Subclass API

Example usage

```
    // continued from previous slide
    RewardCard rc1 = new RewardCard(704, “Paul”);




  }
}
```

Subclass API

Example usage

```
// continued from previous slide
RewardCard rc1 = new RewardCard(704, “Paul”);
output.println(rc1.toString()); // access RewardCard




 }
}
```

95

Subclass API

Example usage

```
// continued from previous slide
RewardCard rc1 = new RewardCard(704, “Paul”);
output.println(rc1.toString()); // access RewardCard
output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.




 }
}
```

96

Subclass API

Example usage

```
    // continued from previous slide
    RewardCard rc1 = new RewardCard(704, “Paul”);
    output.println(rc1.toString()); // access RewardCard
    output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
    output.println(“name: ” + rc1.getName()); // access CreditCard
```

```
  }
}
```

97

Subclass API

Example usage

```
    // continued from previous slide
    RewardCard rc1 = new RewardCard(704, “Paul”);
    output.println(rc1.toString()); // access RewardCard
    output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
    output.println(“name: ” + rc1.getName()); // access CreditCard
    output.println(“bal: ” + rc1.getBalance()); // access Cred.
```

```
  }
}
```

98

Subclass API

Example usage

```
// continued from previous slide
RewardCard rc1 = new RewardCard(704, “Paul”);
output.println(rc1.toString()); // access RewardCard
output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
output.println(“name: ” + rc1.getName()); // access CreditCard
output.println(“bal: ” + rc1.getBalance()); // access Cred.
rc1.charge(500.0); // access Rew.
```

```
 }
}
```

99

Subclass API

Example usage

```
// continued from previous slide
RewardCard rc1 = new RewardCard(704, “Paul”);
output.println(rc1.toString()); // access RewardCard
output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
output.println(“name: ” + rc1.getName()); // access CreditCard
output.println(“bal: ” + rc1.getBalance()); // access Cred.
rc1.charge(500.0); // access Rew.
rc1.pay(50.0); // access Cred.
```

```
 }
}
```

100

Subclass API

Example usage

```
// continued from previous slide
RewardCard rc1 = new RewardCard(704, “Paul”);
output.println(rc1.toString()); // access RewardCard
output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
output.println(“name: ” + rc1.getName()); // access CreditCard
output.println(“bal: ” + rc1.getBalance()); // access Cred.
rc1.charge(500.0); // access Rew.
rc1.pay(50.0); // access Cred.
output.println(“bal: “ + rc1.getBalance()); // access Cred.




 }
}
```

101

Subclass API

Example usage

```
// continued from previous slide
RewardCard rc1 = new RewardCard(704, “Paul”);
output.println(rc1.toString()); // access RewardCard
output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
output.println(“name: ” + rc1.getName()); // access CreditCard
output.println(“bal: ” + rc1.getBalance()); // access Cred.
rc1.charge(500.0); // access Rew.
rc1.pay(50.0); // access Cred.
output.println(“bal: “ + rc1.getBalance()); // access Cred.
output.println(“reward points: ” + rc1.getPointBalance()); //access Rew.




 }
}
```

102

Subclass API

Example usage

```
    // continued from previous slide
    RewardCard rc1 = new RewardCard(704, “Paul”);
    output.println(rc1.toString()); // access RewardCard
    output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
    output.println(“name: ” + rc1.getName()); // access CreditCard
    output.println(“bal: ” + rc1.getBalance()); // access Cred.
    rc1.charge(500.0); // access Rew.
    rc1.pay(50.0); // access Cred.
    output.println(“bal: “ + rc1.getBalance()); // access Cred.
    output.println(“reward points: ” + rc1.getPointBalance()); //access Rew.
    rc1.redeem(5); // access Rew.



  }
}
```

103

Subclass API

Example usage

```
    // continued from previous slide
    RewardCard rc1 = new RewardCard(704, “Paul”);
    output.println(rc1.toString()); // access RewardCard
    output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
    output.println(“name: ” + rc1.getName()); // access CreditCard
    output.println(“bal: ” + rc1.getBalance()); // access Cred.
    rc1.charge(500.0); // access Rew.
    rc1.pay(50.0); // access Cred.
    output.println(“bal: “ + rc1.getBalance()); // access Cred.
    output.println(“reward points: ” + rc1.getPointBalance()); //access Rew.
    rc1.redeem(5); // access Rew.
    output.println(“reward points: ” + rc1.getPointBalance()); //access Rew.



  }
}
```

104

Subclass API

Example usage

```
    // continued from previous slide
    RewardCard rc1 = new RewardCard(704, “Paul”);
    output.println(rc1.toString()); // access RewardCard
    output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
    output.println(“name: ” + rc1.getName()); // access CreditCard
    output.println(“bal: ” + rc1.getBalance()); // access Cred.
    rc1.charge(500.0); // access Rew.
    rc1.pay(50.0); // access Cred.
    output.println(“bal: “ + rc1.getBalance()); // access Cred.
    output.println(“reward points: ” + rc1.getPointBalance()); //access Rew.
    rc1.redeem(5); // access Rew.
    output.println(“reward points: ” + rc1.getPointBalance()); //access Rew.
    rc1.credit(50.0); // access Rew.

  }
}
```

105

Subclass API

Example usage

```
    // continued from previous slide
    RewardCard rc1 = new RewardCard(704, “Paul”);
    output.println(rc1.toString()); // access RewardCard
    output.println(“credit limit: “ + rc1.DEFAULT_LIMIT); // access Rew.
    output.println(“name: ” + rc1.getName()); // access CreditCard
    output.println(“bal: ” + rc1.getBalance()); // access Cred.
    rc1.charge(500.0); // access Rew.
    rc1.pay(50.0); // access Cred.
    output.println(“bal: “ + rc1.getBalance()); // access Cred.
    output.println(“reward points: ” + rc1.getPointBalance()); //access Rew.
    rc1.redeem(5); // access Rew.
    output.println(“reward points: ” + rc1.getPointBalance()); //access Rew.
    rc1.credit(50.0); // access Rew.
    output.println(rc1.toString()); // access Rew.
  }
}
```

106

Outline

- Motivation
- Subclass API
- **Substitutability and Polymorphism**
- Abstract classes and interfaces
- The class Object

107

Substitutability and polymorphism

The is-a relationship (and promotion)

- Every object of the subclass is also a superclass object
  - For example, every Undergrad is-a Student
  - For example, every RewardCard is-a CreditCard

108

Substitutability and Polymorphism

The is-a relationship (and promotion)

- Every object of the subclass is also a superclass object
  - For example, every Undergrad is-a Student
  - For example, every RewardCard is-a CreditCard
- When a superclass reference is expected, a subclass reference will be accepted as well.

109

Substitutability and Polymorphism

The is-a relationship (and promotion)

- Every object of the subclass is also a superclass object
  - For example, every Undergrad is-a Student
  - For example, every RewardCard is-a CreditCard
- When a superclass reference is expected, a subclass reference will be accepted as well.
- Here are a few of examples

```
CreditCard cc = new RewardCard(); // cc declared Credit
output.println("I am a string."); // println expects an Object
```

110

Substitutability and Polymorphism

The is-a relationship (and promotion)

- Every object of the subclass is also a superclass object
  - For example, every Undergrad is-a Student
  - For example, every RewardCard is-a CreditCard
- When a superclass reference is expected, a subclass reference will be accepted as well.
- Here are a few of examples

```
CreditCard cc = new RewardCard(); // cc declared Credit
output.println("I am a string."); // println expects an Object
```

- This is analogous to automatic promotion among primitive types.

111

Substitutability and Polymorphism

The is-a relationship (and demotion)

- Demotion between object types (within a hierarchy) requires manual casting
- Again, as among primitive types.

112

Substitutability and Polymorphism

The is-a relationship (and demotion)

- Demotion between object types (within a hierarchy) requires manual casting
- Again, as among primitive types.
- It is advisable to test via instanceof prior to casting

```
CreditCard cc = new RewardCard( );
...
if (cc instanceof RewardCard)
  RewardCard rc = (RewardCard) cc;
```

113

Substitutability and Polymorphism

The is-a relationship (and demotion)

- Demotion between object types (within a hierarchy) requires manual casting
- Again, as among primitive types.
- It is advisable to test via instanceof prior to casting

```
CreditCard cc = new RewardCard( );
...
if (cc instanceof RewardCard)
  RewardCard rc = (RewardCard) cc;
```

- Attempt of an inappropriate demotion yields a run-time error.

114

Substitutability and Polymorphism

The substitutability principle

- The sort of automatic promotion between types within a hierarchy that we have seen, e.g.,…

```
CreditCard cc = new RewardCard( );
```

- …is enabled at the level of the compiler by the so called…
- Substitutability principle: When a superclass is expected, a subclass is accepted.

115

Substitutability and Polymorphism

The substitutability principle

- The sort of automatic promotion between types within a hierarchy that we have seen, e.g.,…

```
CreditCard cc = new RewardCard( );
```

- …is enabled at the level of the compiler by the so called…
- Substitutability principle: When a superclass is expected, a subclass is accepted.
- Remark: For the most part this principle is perfectly reasonable; the subclass has *at least* the representational power of the superclass (although perhaps in a somewhat different form through overriding and shadowing).

116

Substitutability and Polymorphism

At run-time…

- …the processor invokes overriden instance methods based on the object type, not the reference type.

117

Substitutability and Polymorphism

At run-time…

- …the processor invokes overridden instance methods based on the object type, not the reference type.
- Example

```
CreditCard cc1 = new CreditCard( );
CreditCard cc2 = new RewardCard( );
```

118

Substitutability and Polymorphism

At run-time…

- …the processor invokes overridden instance methods based on the object type, not the reference type.
- Example

```
CreditCard cc1 = new CreditCard( );
CreditCard cc2 = new RewardCard( );
output.println(cc1); // invokes toString of CreditCard
```

119

Substitutability and Polymorphism

At run-time…

- …the processor invokes overridden instance methods based on the object type, not the reference type.
- Example

```
CreditCard cc1 = new CreditCard( );
CreditCard cc2 = new RewardCard( );
output.println(cc1); // invokes toString of CreditCard
output.println(cc2); // invokes toString of RewardCard
```

120

Substitutability and Polymorphism

At run-time…

- …the processor invokes overridden instance methods based on the object type, not the reference type.
- Example

```
CreditCard cc1 = new CreditCard( );
CreditCard cc2 = new RewardCard( );
output.println(cc1); // invokes toString of CreditCard
output.println(cc2); // invokes toString of RewardCard
Object obj1 = cc1;
Object obj2 = cc2;
```

121

Substitutability and Polymorphism

At run-time…

- …the processor invokes overridden instance methods based on the object type, not the reference type.
- Example

```
CreditCard cc1 = new CreditCard( );
CreditCard cc2 = new RewardCard( );
output.println(cc1); // invokes toString of CreditCard
output.println(cc2); // invokes toString of RewardCard
Object obj1 = cc1;
Object obj2 = cc2;
output.println(obj1); // invokes toString of CreditCard
output.println(obj2); // invokes toString of RewardCard
```

122

Substitutability and Polymorphism

Early vs. late binding

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At compile time (early binding) it is okay to supply small when big is expected

123

Substitutability and Polymorphism

Early vs. late binding

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At compile time (early binding) it is okay to supply small when big is expected
  - **Assignment:** big = small;

124

Substitutability and Polymorphism

Early vs. late binding

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At compile time (early binding) it is okay to supply small when big is expected
  - **Assignment:** big = small;
    CreditCard cc = new RewardCard(1, “Smith”);

125

Substitutability and Polymorphism

Early vs. late binding

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At compile time (early binding) it is okay to supply small when big is expected
  - **Assignment:** big = small;
    CreditCard cc = new RewardCard(1, “Smith”);
  - **Parameter passing:** pass a small argument to method with big parameters

126

Substitutability and Polymorphism

Early vs. late binding

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At compile time (early binding) it is okay to supply small when big is expected
  - **Assignment:** big = small;

    CreditCard cc = new RewardCard(1, “Smith”);
  - **Parameter passing:** pass a small argument to method with big parameters

    GlobalCredit gcc = new GlobalCredit();

    gcc.add(new RewardCard(1, “Smith”));

127

Substitutability and Polymorphism

Early vs. late binding

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At compile time (early binding) it is okay to supply small when big is expected
  - **Assignment:** big = small;

    CreditCard cc = new RewardCard(1, “Smith”);
  - **Parameter passing:** pass a small argument to method with big parameters

    GlobalCredit gcc = new GlobalCredit();

    gcc.add(new RewardCard(1, “Smith”));
  - **Return:** Return a small in a big return method.

128

Substitutability and Polymorphism

Early vs. late binding

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At compile time (early binding) it is okay to supply small when big is expected
  - **Assignment:** big = small;
    CreditCard cc = new RewardCard(1, “Smith”);
  - **Parameter passing:** pass a small argument to method with big parameters
    GlobalCredit gcc = new GlobalCredit();
    gcc.add(new RewardCard(1, “Smith”));
  - **Return:** Return a small in a big return method.
    cc = gcc.getFirst(); // hypothetical getFirst() method

129

Substitutability and Polymorphism

Early vs. late binding (Cont.)

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At run-time (late binding) it is okay to reverse all of the above

Substitutability and Polymorphism

Early vs. late binding (Cont.)

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At run-time (late binding) it is okay to reverse all of the above
  - via use of the *potentially dangerous* cast (small) big

```
RewardCard rc = (RewardCard) cc;
```

Substitutability and Polymorphism

Early vs. late binding (Cont.)

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At run-time (late binding) it is okay to reverse all of the above
  - via use of the *potentially dangerous* cast (small) big

```
RewardCard rc = (RewardCard) cc;
```

  - This *only works if* big actually points to a small object at run-time, e.g., prior to cast had something equivalent to

```
CreditCard cc = new RewardCard();
```

Substitutability and Polymorphism

Early vs. late binding (Cont.)

- Suppose we are given two references, big and small, to a superclass and its subclass.
- At run-time (late binding) it is okay to reverse all of the above
  - via use of the *potentially dangerous* cast (small) big

    ```
    RewardCard rc = (RewardCard) cc;
    ```
  - This *only works if* big actually points to a small object at run-time, e.g., prior to cast had something equivalent to

    ```
    CreditCard cc = new RewardCard();
    ```
  - Make use of instanceof to avoid having a run-time error (an exception thrown), e.g.,

    ```
    if (cc instanceof RewardCard)
      RewardCard rc = (RewardCard) cc;
    ```

133

Substitutability and Polymorphism

Reference resolution

- Let r be a reference to an object o.

134

Substitutability and Polymorphism

Reference resolution

- Let r be a reference to an object o.
  - Remark: Due to substitutability, e.g.,

    CreditCard cc = new RewardCard();

    the class of the reference (e.g., cc has class CreditCard) is not necessarily the class of the actual object instance in memory (e.g., RewardCard).

135

Substitutability and Polymorphism

Reference resolution

- Let r be a reference to an object o.
  - Remark: Due to substitutability, e.g.,

    CreditCard cc = new RewardCard();

    the class of the reference (e.g., cc has class CreditCard) is not necessarily the class of the actual object instance in memory (e.g., RewardCard).
- Let f be a feature, i.e., a method or attribute.

136

Substitutability and Polymorphism

Reference resolution

- Let r be a reference to an object o.
  - Remark: Due to substitutability, e.g.,

    CreditCard cc = new RewardCard();

    the class of the reference (e.g., cc has class CreditCard) is not necessarily the class of the actual object instance in memory (e.g., RewardCard).
- Let f be a feature, i.e., a method or attribute.
- Problem: Given r.f, what is the target class used to realize the desired computation?

137

Substitutability and Polymorphism

Reference resolution

- Let r be a reference to an object o.
  - Remark: Due to substitutability, e.g.,

    CreditCard cc = new RewardCard();

    the class of the reference (e.g., cc has class CreditCard) is not necessarily the class of the actual object instance in memory (e.g., RewardCard).
- Let f be a feature, i.e., a method or attribute.
- Problem: Given r.f, what is the target class used to realize the desired computation?
  - Is it that of the reference (e.g., CreditCard)?

    or
  - Is it that of the object instance (e.g., RewardCard)?

138

Substitutability and Polymorphism

Reference resolution

- Let r be a reference to an object o.
- Let f be a feature, i.e., a method or attribute.
- Problem: Given r.f, what is the target class used to realize the desired computation?
- Solution (in two phases):

Substitutability and Polymorphism

Reference resolution

- Let r be a reference to an object o.
- Let f be a feature, i.e., a method or attribute.
- Problem: Given r.f, what is the target class used to realize the desired computation?
- Solution (in two phases):
  - **Early binding** solution (realized at compile time by compiler):
    target class = class of r

    regardless of the class of the actual object.

Substitutability and Polymorphism

Reference resolution

- Let r be a reference to an object o.
- Let f be a feature, i.e., a method or attribute.
- Problem: Given r.f, what is the target class used to realize the desired computation?
- Solution (in two phases):
  - **Early binding** solution (realized at compile time by compiler):
    target class = class of r
    regardless of the class of the actual object.
  - **Late binding** solution (realized at run-time by the virtual machine):
    if (f is not an overriding instance method)
        late binding target class = early binding target class
    else
        late binding target class = class of o

Substitutability and Polymorphism

Reference resolution

- Let r be a reference to an object o.
- Let f be a feature, i.e., a method or attribute.
- Problem: Given r.f, what is the target class used to realize the desired computation?
- Solution (in two phases):
  - **Early binding** solution (realized at compile time by compiler):
    target class = class of r
    regardless of the class of the actual object.
  - **Late binding** solution (realized at run-time by the virtual machine):
    if (f is not an overriding instance method)
        late binding target class = early binding target class
    else
        late binding target class = class of o

The executed computation is in terms of the late binding result.

142

Substitutability and Polymorphism

Reference resolution

- Example

```
CreditCard cc = new RewardCard();
```

Substitutability and Polymorphism

Reference resolution

- Example

```
CreditCard cc = new RewardCard();
```

- Based on the approach to reference resolution, early binding (based on the reference) is upheld for
  - Attributes of all types (static/class as well as non-static/instance); even shadowed fields will not be blocked.

```
double d = cc.DEFAULT_RATE; // accesses CreditCard
```

Substitutability and Polymorphism

Reference resolution

- Example

```
CreditCard cc = new RewardCard();
```

- Based on the approach to reference resolution, early binding (based on the reference) is upheld for
  - Attributes of all types (static/class as well as non-static/instance); even shadowed fields will not be blocked.

```
double d = cc.DEFAULT_RATE; // accesses CreditCard
```

  - Static/class methods, even if invoked via the reference (as opposed to the class name).

```
cc.hypotheticalStaticMethod(); // accesses CreditCard
```

Substitutability and Polymorphism

Reference resolution

- Example

```
CreditCard cc = new RewardCard();
```

- Based on the approach to reference resolution, early binding (based on the reference) is upheld for
  - Attributes of all types (static/class as well as non-static/instance); even shadowed fields will not be blocked.

```
double d = cc.DEFAULT_RATE; // accesses CreditCard
```

  - Static/class methods, even if invoked via the reference (as opposed to the class name).

```
cc.hypotheticalStaticMethod(); // invokes CreditCard
```

  - Non-overridden instance methods.

```
cc.pay(50.0); // invokes CreditCard
```

Substitutability and Polymorphism

Reference resolution

- Example

  ```
  CreditCard cc = new RewardCard();
  ```

- Based on the approach to reference resolution, early binding (based on the reference) is upheld for
  - Attributes of all types (static/class as well as non-static/instance); even shadowed fields will not be blocked.

    ```
    double d = cc.DEFAULT_RATE; // accesses CreditCard
    ```

  - Static/class methods, even if invoked via the reference (as opposed to the class name).

    ```
    cc.hypotheticalStaticMethod(); // invokes CreditCard
    ```

  - Non-overridden instance methods.

    ```
    cc.pay(50.0); // invokes CreditCard
    ```

  Late binding only changes matters for overridden instance methods.

  ```
  cc.charge(200.00); // invokes RewardCard
  ```

147

Substitutability and Polymorphism

The polymorphism principle

- In Java, instance method calls are always determined by the type of the actual object, not the type of the object reference.

148

Substitutability and Polymorphism

The polymorphism principle

- In Java, instance method calls are always determined by the type of the actual object, not the type of the object reference.
- The principle that the actual type of the object determines the method to be called is polymorphism.
  - Many forms; same name.
  - The same computation works for objects of many shapes.
  - It adapts itself to the nature of the objects.

149

Substitutability and Polymorphism

The polymorphism principle

- In Java, instance method calls are always determined by the type of the actual object, not the type of the object reference.
- The principle that the actual type of the object determines the method to be called is polymorphism.
  - Many forms; same name.
  - The same computation works for objects of many shapes.
  - It adapts itself to the nature of the objects.
- Early binding pertains to compile time polymorphism.

150

Substitutability and Polymorphism

The polymorphism principle

- In Java, instance method calls are always determined by the type of the actual object, not the type of the object reference.
- The principle that the actual type of the object determines the method to be called is polymorphism.
  - Many forms; same name.
  - The same computation works for objects of many shapes.
  - It adapts itself to the nature of the objects.
- Early binding pertains to compile time polymorphism.
- Late binding pertains to run-time polymorphism.

151

Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {
```

152

Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {  GlobalCredit yc = new GlobalCredit(“York Credit”);
```


Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {  GlobalCredit yc = new GlobalCredit(“York Credit”);
     output.println(yc.toString());
```

Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {  GlobalCredit yc = new GlobalCredit("York Credit");
     output.println(yc.toString());
     CreditCard cc1 = new CreditCard(703,"John",2000.0);
```

155

Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {  GlobalCredit yc = new GlobalCredit("York Credit");
     output.println(yc.toString());
     CreditCard cc1 = new CreditCard(703,"John",2000.0);
     yc.add(cc1); // expects CC ref., receives and accepts CC ref
```

156

Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {  GlobalCredit yc = new GlobalCredit("York Credit");
     output.println(yc.toString());
     CreditCard cc1 = new CreditCard(703,"John",2000.0);
     yc.add(cc1); // expects CC ref., receives and accepts CC ref
     output.println(yc.toString());
```

157

Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {  GlobalCredit yc = new GlobalCredit("York Credit");
     output.println(yc.toString());
     CreditCard cc1 = new CreditCard(703,"John",2000.0);
     yc.add(cc1); // expects CC ref., receives and accepts CC ref
     output.println(yc.toString());
     CreditCard cc2 = new RewardCard(704,"Paul",1000.0);
     yc.add(cc2); // expects CC ref., receives and accepts CC ref
```

158

Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {  GlobalCredit yc = new GlobalCredit("York Credit");
     output.println(yc.toString());
     CreditCard cc1 = new CreditCard(703,"John",2000.0);
     yc.add(cc1); // expects CC ref., receives and accepts CC ref
     output.println(yc.toString());
     CreditCard cc2 = new RewardCard(704,"Paul",1000.0);
     yc.add(cc2); // expects CC ref., receives and accepts CC ref
     RewardCard rc1 = new RewardCard(705,"Jane",2500.0);
     yc.add(rc1); // expects CC ref., receives and accepts RC ref
```

159

Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {  GlobalCredit yc = new GlobalCredit("York Credit");
     output.println(yc.toString());
     CreditCard cc1 = new CreditCard(703,"John",2000.0);
     yc.add(cc1); // expects CC ref., receives and accepts CC ref
     output.println(yc.toString());
     CreditCard cc2 = new RewardCard(704,"Paul",1000.0);
     yc.add(cc2); // expects CC ref., receives and accepts CC ref
     RewardCard rc1 = new RewardCard(705,"Jane",2500.0);
     yc.add(rc1); // expects CC ref., receives and accepts RC ref
     output.println(yc.toString());
```

160

Substitutability and Polymorphism

Example usage

```
// assume the usual
import type.lib.*;
public class GlobalCreditEg
{ public static void main(String[ ] args)
  {  GlobalCredit yc = new GlobalCredit("York Credit");
     output.println(yc.toString());
     CreditCard cc1 = new CreditCard(703,"John",2000.0);
     yc.add(cc1); // expects CC ref., receives and accepts CC ref
     output.println(yc.toString());
     CreditCard cc2 = new RewardCard(704,"Paul",1000.0);
     yc.add(cc2); // expects CC ref., receives and accepts CC ref
     RewardCard rc1 = new RewardCard(705,"Jane",2500.0);
     yc.add(rc1); // expects CC ref., receives and accepts RC ref
     output.println(yc.toString());
     // continued on next slide
```

161

Substitutability and Polymorphism

Example usage

```
     // continued from previous slide
```

162

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
boolean res;
res = cc2.charge(500.0); // Reward
```

163

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
```

164

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
res = cc1.charge(600.0); // Credit
output.println("charging 600.0 to 703 is " + res); // true
```

165

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
res = cc1.charge(600.0); // Credit
output.println("charging 600.0 to 703 is " + res); // true
for (CreditCard cc : yc)
```

166

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
res = cc1.charge(600.0); // Credit
output.println("charging 600.0 to 703 is " + res); // true
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
```

167

Substitutability and Polymorphism

Example usage

```
// continued from prev
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
res = cc1.charge(600.0); // Credit
output.println("charging 600.0 to 703 is " + res); // true
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
```

```
RWRD [NO=000704-7, Balance=500.00, Points=25]
RWRD [NO=000705-6, Balance=0.00, Points=0]
CARD [NO=000703-8, Balance=600.00]
```

168

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
res = cc1.charge(600.0); // Credit
output.println("charging 600.0 to 703 is " + res); // true
for (CreditCard cc : yc)
      output.println(cc.toString()); // depends on object type
res = cc2.charge(800.0); // Reward
output.println("charging 800.0 to 704 is " + res); // false
```

169

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
res = cc1.charge(600.0); // Credit
output.println("charging 600.0 to 703 is " + res); // true
for (CreditCard cc : yc)
      output.println(cc.toString()); // depends on object type
res = cc2.charge(800.0); // Reward
output.println("charging 800.0 to 704 is " + res); // false
res = cc2.charge(400.0); // Reward
output.println("charging 400.0 to 704 is " + res); // true
```

170

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
res = cc1.charge(600.0); // Credit
output.println("charging 600.0 to 703 is " + res); // true
for (CreditCard cc : yc)
      output.println(cc.toString()); // depends on object type
res = cc2.charge(800.0); // Reward
output.println("charging 800.0 to 704 is " + res); // false
res = cc2.charge(400.0); // Reward
output.println("charging 400.0 to 704 is " + res); // true
for (CreditCard cc : yc)
      output.println(cc.toString()); // depends on object type
```

171

Substitutability and Polymorphism

Example usage

```
// continued from prev
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
res = cc1.charge(600.0); // Credit
output.println("charging 600.0 to 703 is " + res); // true
for (CreditCard cc : yc)
      output.println(cc.toString()); // depends on object type
res = cc2.charge(800.0); // Reward
output.println("charging 800.0 to 704 is " + res); // false
res = cc2.charge(400.0); // Reward
output.println("charging 400.0 to 704 is " + res); // true
for (CreditCard cc : yc)
      output.println(cc.toString()); // depends on object type
```

```
RWRD [NO=000704-7, Balance=900.00, Points=45]
RWRD [NO=000705-6, Balance=0.00, Points=0]
CARD  [NO=000703-8, Balance=600.00]
```

172

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
boolean res;
res = cc2.charge(500.0); // Reward
output.println("charging 500.0 to 704 is " + res); // true
res = cc1.charge(600.0); // Credit
output.println("charging 600.0 to 703 is " + res); // true
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
res = cc2.charge(800.0); // Reward
output.println("charging 800.0 to 704 is " + res); // false
res = cc2.charge(400.0); // Reward
output.println("charging 400.0 to 704 is " + res); // true
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
// continued on next page
```


Substitutability and Polymorphism

Example usage

```
// continued from previous slide
```

```
} }
```

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
cc1.pay(200.0); // CC
output.println("Pay 200 on 703.");
```

} }

175

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
cc1.pay(200.0); // CC
output.println("Pay 200 on 703.");
((RewardCard) cc2).redeem(10); // risky, but we know cc2 is RC
output.println("redeem 10 points on 704.");
```

} }

176

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
cc1.pay(200.0); // CC
output.println("Pay 200 on 703.");
((RewardCard) cc2).redeem(10); // risky, but we know cc2 is RC
output.println("redeem 10 points on 704.");
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
```

} }

Substitutability and Polymorphism

Example usage

```
// continued from prev
cc1.pay(200.0); // CC
output.println("Pay 200 on 703.");
((RewardCard) cc2).redeem(10); // risky, but we know cc2 is RC
output.println("redeem 10 points on 704.");
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
```

```
RWRD [NO=000704-7, Balance=900.00, Points=35]
RWRD [NO=000705-6, Balance=0.00, Points=0]
CARD [NO=000703-8, Balance=400.00]
```

} }

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
cc1.pay(200.0); // CC
output.println("Pay 200 on 703.");
((RewardCard) cc2).redeem(10); // risky, but we know cc2 is RC
output.println("redeem 10 points on 704.");
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
res = rc1.charge(800.0); // RC
output.println("charging 800.0 to 705 is " + res); // true
```

} }

179

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
cc1.pay(200.0); // CC
output.println("Pay 200 on 703.");
((RewardCard) cc2).redeem(10); // risky, but we know cc2 is RC
output.println("redeem 10 points on 704.");
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
res = rc1.charge(800.0); // RC
output.println("charging 800.0 to 705 is " + res); // true
for (CreditCard cc : yc)
{



}
```

} }

180

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
cc1.pay(200.0); // CC
output.println("Pay 200 on 703.");
((RewardCard) cc2).redeem(10); // risky, but we know cc2 is RC
output.println("redeem 10 points on 704.");
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
res = rc1.charge(800.0); // RC
output.println("charging 800.0 to 705 is " + res); // true
for (CreditCard cc : yc)
{  if (cc instanceof RewardCard) // true for cc2 and rc1
   {

   }
}
} }
```

181

Substitutability and Polymorphism

Example usage

```
// continued from previous slide
cc1.pay(200.0); // CC
output.println("Pay 200 on 703.");
((RewardCard) cc2).redeem(10); // risky, but we know cc2 is RC
output.println("redeem 10 points on 704.");
for (CreditCard cc : yc)
        output.println(cc.toString()); // depends on object type
res = rc1.charge(800.0); // RC
output.println("charging 800.0 to 705 is " + res); // true
for (CreditCard cc : yc)
{  if (cc instanceof RewardCard) // true for cc2 and rc1
   {  RewardCard rc = (RewardCard) cc;
      output.println(rc.getNumber() + " " + rc.getPointBalance());
   }
}
} }
```

182

Substitutability and Polymorphism

Example usage

```
    // continued from previous slide
    cc1.pay(200.0); // CC
    output.println("Pay 200 on 703.");
    ((RewardCard) cc2).redeem(10); // risky, but we know cc2 is RC
    output.println("redeem 10 points on 704.");
    for (CreditCard cc : yc)
          output.println(cc.toString()); // depends on object type
    res = rc1.charge(800.0); // RC
    output.println("charging 800.0 to 705 is " + res); // true
    for (CreditCard cc : yc)
    {  if (cc instanceof RewardCard) // true for cc2 and rc1
      {  RewardCard rc = (RewardCard) cc;
        output.println(rc.getNumber() + " " + rc.getPointBalance());
      }
    }
  } }
```

000704-7 35
000705-6 40

183

Outline

- Motivation
- Subclass API
- Substitutability and Polymorphism
- **Abstract classes and interfaces**
- The class Object

184

Abstract classes and interfaces

A challenge

- Designing classes hierarchically has great power in allowing subclasses to inherit from superclasses.
- In particular, when one class contains a subset of the features of another, then it is natural to specify them as subclass in superclass in an inheritance chain.

185

Abstract classes and interfaces

A challenge

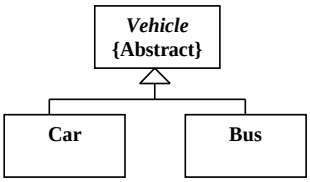
- Designing classes hierarchically has great power in allowing subclasses to inherit from superclasses.
- In particular, when one class contains a subset of the features of another, then it is natural to specify them as subclass in superclass in an inheritance chain.
- In some situations, however, two classes share some features, but each contains features not in the other.
- Java provides two ways to deal with such situations.
  - Abstract classes
  - Interfaces

186

Abstract classes and interfaces

Abstract classes

- Given two classes that contain some common features, create an artificial third class and designate it as the superclass of the other two.
- The artificial class is called an *abstract* class because it does not encapsulate an actual object.
- Still, an inheritance hierarchy is defined with the abstract class at the root.

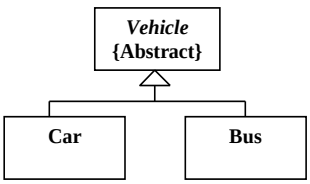


187

Abstract classes and interfaces

Abstract classes

- Given two classes that contain some common features, create an artificial third class and designate it as the superclass of the other two.
- The artificial class is called an *abstract* class because it does not encapsulate an actual object.
- Still, an inheritance hierarchy is defined with the abstract class at the root.



- An abstract class is recognizable in its API header because the word abstract is added.

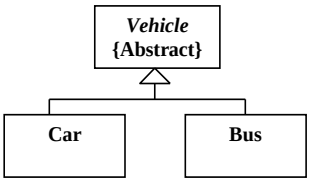
```
public abstract class Vehicle
```

188

Abstract classes and interfaces

Abstract classes

- Given two classes that contain some common features, create an artificial third class and designate it as the superclass of the other two.
- The artificial class is called an *abstract* class because it does not encapsulate an actual object.
- Still, an inheritance hierarchy is defined with the abstract class at the root.



In UML, an abstract class has its name in italics and the word Abstract beneath it in curly brackets.

- An abstract class is recognizable in its API header because the word abstract is added.

```
public abstract class Vehicle
```

189

Abstract classes and interfaces

Abstract class usage

- An abstract class cannot be instantiated.
- To use an abstract class, you must obtain an instance from one of its subclasses.
- There are two standard ways to proceed.

Abstract classes and interfaces

Abstract class usage

- An abstract class cannot be instantiated.
- To use an abstract class, you must obtain an instance from one of its subclasses.
- There are two standard ways to proceed.

1. Factory method: Find a method that gives you an instance. For example, the abstract class Vehicle has a method

```
public static Car createCar()
```

Then an instance of vehicle can be created as Vehicle

```
Vehicle myCar = Vehicle.createCar();
```

Abstract classes and interfaces

Abstract class usage

- An abstract class cannot be instantiated.
- To use an abstract class, you must obtain an instance from one of its subclasses.
- There are two standard ways to proceed.

1. Factory method: Find a method that gives you an instance. For example, the abstract class Vehicle has a method

```
public static Car createCar()
```

Then an instance of vehicle can be created as Vehicle

```
Vehicle myCar = Vehicle.createCar();
```

2. Subclass constructor: Find a subclass and use its constructor.

```
Vehicle myCar = new Car();
```

192

Abstract classes and interfaces

Interfaces

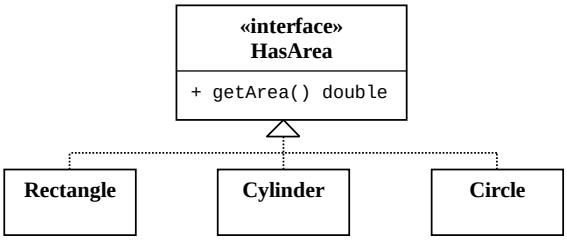
- In some cases it might be desirable for a class to inherit features from more than one class.
- However, at least in Java, multiple inheritance is not allowed.
- An interface defines a group of methods that must be define by its implementing classes.
- The implementing class does not inherit the methods in full, only their headers (i.e., the way to interface to them).
- The key: A class is allowed to implement multiple interfaces.

193

Abstract classes and interfaces

Interfaces

- As an example, suppose you wanted to abstract the notion that a variety of shapes have area.



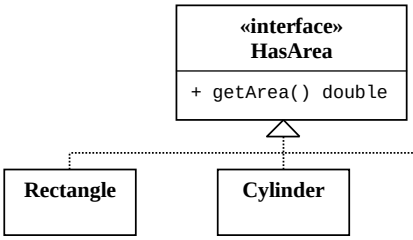
- Notice that each implementing class must implement getArea(). Indeed, the necessary operations are shape dependent.

194

Abstract classes and interfaces

Interfaces

- As an example, suppose you wanted to abstract the notion that a variety of shapes have area.



In UML, an interface is indicated via <<interface>> above its name.

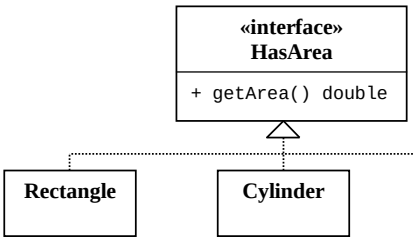
- Notice that each implementing class must implement getArea(). Indeed, the necessary operations are shape dependent.

195

Abstract classes and interfaces

Interfaces

- As an example, suppose you wanted to abstract the notion that a variety of shapes have area.



In UML, an interface is indicated via <<interface>> above its name.

- Notice that each implementing class must implement getArea(). Indeed, the necessary operations are shape dependent.
- To gain access, we appeal to an implementing class, e.g.,

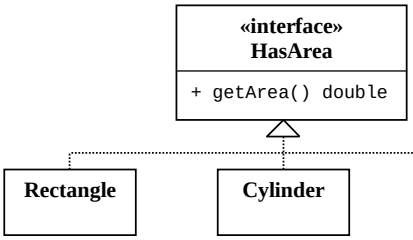
HasArea shape = new Rectangle(3, 9);

196

Abstract classes and interfaces

Interfaces

- As an example, suppose you wanted to abstract the notion that a variety of shapes have area.



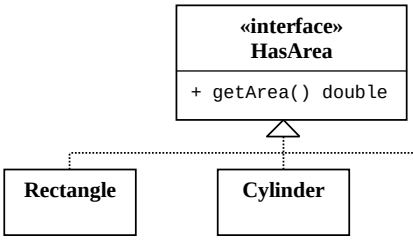
In UML, an interface is indicated via <<interface>> above its name.

- Remark: Since we are dealing with interfaces, each of the above classes might also implement additional interfaces.

Abstract classes and interfaces

Interfaces

- As an example, suppose you wanted to abstract the notion that a variety of shapes have area.



In UML, an interface is indicated via <<interface>> above its name.

- Remark: Since we are dealing with interfaces, each of the above classes might also implement additional interfaces.

- As examples:
  - Rectangle and Cylinder might also implement an interface HasHeight
  - Cylinder and Circle might also implement an interface HasRadius

Abstract classes and interfaces

Summary

- Designing classes hierarchically has great power in allowing subclasses to inherit from superclasses.
- In some situations, however, two classes share some features, but each contains features not in the other.
- Java provides two ways to deal with such situations.
  - Abstract classes
  - Interfaces

199

Outline

- Motivation
- Subclass API
- Substitutability and Polymorphism
- Abstract classes and interfaces
- **The class Object**

200

The class Object

The cosmic superclass

- In Java, all objects are direct or indirect subclasses of the Object class.

The class Object

The cosmic superclass

- In Java, all objects are direct or indirect subclasses of the Object class.
- They inherit from it a number of methods
  - toString
  - equals
  - clone
  - and more…

202

The class Object

The cosmic superclass

- All objects are direct or indirect subclasses of the Object class.
- They inherit from it a number of methods
  - toString
  - equals
  - clone
  - and more…
- As with other inherited methods, it is critical to consider how they work...
- …and override them, as needed.

203

The class Object

The cosmic superclass

- In Java, all objects are direct or indirect subclasses of the Object class.
- They inherit from it a number of methods
  - toString
  - equals
  - clone
  - and more…
- As with other inherited methods, it is critical to consider how they work...
- …and override them, as needed.
- For example, the toString of RewardCard is an override of the toString inherited from CreditCard, which in turn is an override of the toString inherited from Object.

204

The class Object

To override or not to override

- **Example:** The Object class clone provides only shallow copying

205

The class Object

To override or not to override

- **Example:** The Object class clone provides only shallow copying
  - If an object contains a reference to another object, then it provides a copy (not a clone) of that object reference.

206

The class Object

To override or not to override

- **Example:** The Object class clone provides only shallow copying
  - If an object contains a reference to another object, then it provides a copy (not a clone) of that object reference.
  - Okay if all attributes are numbers, truth values or strings (immutable objects)
  - Suspect if (mutable) objects are involved.

207

The class Object

To override or not to override

- **Example:** The Object class clone provides only shallow copying
  - If an object contains a reference to another object, then it provides a copy (not a clone) of that object reference.
  - Okay if all attributes are numbers, truth values or strings (immutable objects)
  - Suspect if (mutable) objects are involved.
- **Example:** The Object class equals defaults to = =

208

The class Object

To override or not to override

- **Example:** The Object class clone provides only shallow copying
  - If an object contains a reference to another object, then it provides a copy (not a clone) of that object reference.
  - Okay if all attributes are numbers, truth values or strings (immutable objects)
  - Suspect if (mutable) objects are involved.
- **Example:** The Object class equals defaults to = =
- **Example:** The Object class toString defaults to the object reference.

209

The class Object

Why bother?

210

The class Object

Why bother?

- **Theory benefit (example):** Use of the Object class enforces an overall hierarchical structure on Java data objects.

211

The class Object

Why bother?

- **Theory benefit (example):** Use of the Object class enforces an overall hierarchical structure on Java data objects.
- **Practical benefit (example):** Use of the Object class allows us to define generic data structuring mechanisms (recall the class Vector, which can hold any Object).

212

The class Object

Recap.

- In Java, all objects are direct or indirect subclasses of the Object class.
- Correspondingly, all classes inherit a number of methods from Object.
  - As with other inherited methods, it is critical to consider how they work and override them, as needed.
- Use of the Object class provides
  - Overall hierarchical structure to Java.
  - Ability to define and use generic data structuring and manipulation.

213

Summary

- **Motivation**
- **Subclass API**
- **Substitutability and Polymorphism**
- **Abstract classes and interfaces**
- **The class Object**

214

Outline

- Motivation
- Subclass API
- Substitutability and Polymorphism
- Abstract classes and interfaces
- The class Object
- **Appendix: Arrays of objects**

215

Arrays of objects

Array elements can be objects

- As one would expect, we can create arrays of objects.
- Let's see an example…

216

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)




}
```

217

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;




}
```

218

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;
    CreditCard[ ] myCards = new CreditCard[MAX_CARDS];




}
```

219

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;
    CreditCard[ ] myCards = new CreditCard[MAX_CARDS];
    myCards[0] = new CreditCard(1, “Wildes”);




}
```

220

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;
    CreditCard[ ] myCards = new CreditCard[MAX_CARDS];
    myCards[0] = new CreditCard(1, “Wildes”);
    myCards[1] = new CreditCard(2, “Wildes”);




}
```

221

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;
    CreditCard[ ] myCards = new CreditCard[MAX_CARDS];
    myCards[0] = new CreditCard(1, “Wildes”);
    myCards[1] = new CreditCard(2, “Wildes”);
    myCards[2] = new RewardCard(3, “Wildes”);




}
```

222

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;
    CreditCard[ ] myCards = new CreditCard[MAX_CARDS];
    myCards[0] = new CreditCard(1, “Wildes”);
    myCards[1] = new CreditCard(2, “Wildes”);
    myCards[2] = new RewardCard(3, “Wildes”);
```

Remark: The principle of substitutability allows a RewardCard instance to be inserted in a CreditCard array.

```
}
```

223

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;
    CreditCard[ ] myCards = new CreditCard[MAX_CARDS];
    myCards[0] = new CreditCard(1, “Wildes”);
    myCards[1] = new CreditCard(2, “Wildes”);
    myCards[2] = new RewardCard(3, “Wildes”);
    output.println(“Cards in my wallet\n”);


  }
}
```

224

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;
    CreditCard[ ] myCards = new CreditCard[MAX_CARDS];
    myCards[0] = new CreditCard(1, “Wildes”);
    myCards[1] = new CreditCard(2, “Wildes”);
    myCards[2] = new RewardCard(3, “Wildes”);
    output.println(“Cards in my wallet\n”);
    for (int j=0; j<myCards.length; j++)

  }
}
```

225

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;
    CreditCard[ ] myCards = new CreditCard[MAX_CARDS];
    myCards[0] = new CreditCard(1, “Wildes”);
    myCards[1] = new CreditCard(2, “Wildes”);
    myCards[2] = new RewardCard(3, “Wildes”);
    output.println(“Cards in my wallet\n”);
    for (int j=0; j<myCards.length; j++)
      output.println(myCards[j].toString());
  }
}
```

Arrays of objects

Example

```
public class ArrayOfCards
{ public static void main(String[ ] args)
  { final int MAX_CARDS = 3;
    CreditCard[ ] myCards = new CreditCard[MAX_CARDS];
    myCards[0] = new CreditCard(1, “Wildes”);
    myCards[1] = new CreditCard(2, “Wildes”);
    myCards[2] = new RewardCard(3, “Wildes”);
    output.println(“Cards in my wallet\n”);
    for (int j=0; j<myCards.length; j++)
      output.println(myCards[j].toString());
  }
}
```

Remark: Polymorphism ensures that the appropriate toString method will be called for each card.

227

Arrays of objects

Recap.

- We can create arrays of objects.
- Declaration and construction exactly analogous to that of arrays of primitive type elements.
- When applicable, the principles of substitutability and polymorphism work exactly as they should.

228