



Introduction to Inheritance

James Brucker

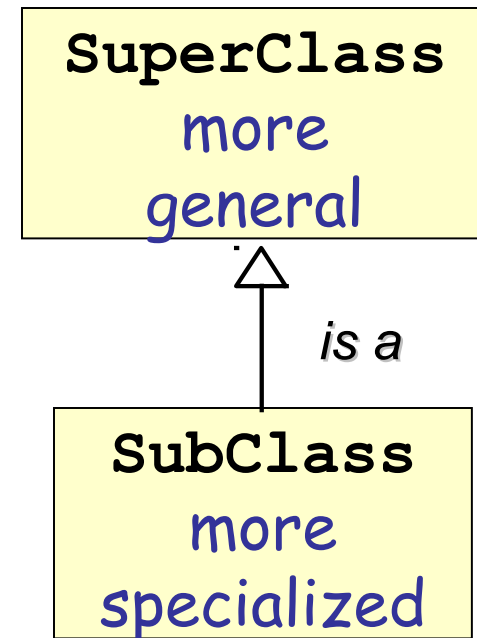
These slides cover only the basics of inheritance.

For deeper understanding, see the textbook and study the programming assignments.

What is Inheritance?

One class incorporates all the attributes and behavior from another class -- it *inherits* these attributes and behavior.

- ❑ A subclass *inherits all* the attributes and behavior of the superclass.
- ❑ It can directly *access* the public & protected members of the superclass.
- ❑ Subclass can *redefine* some inherited behavior, or add new attributes and behavior.



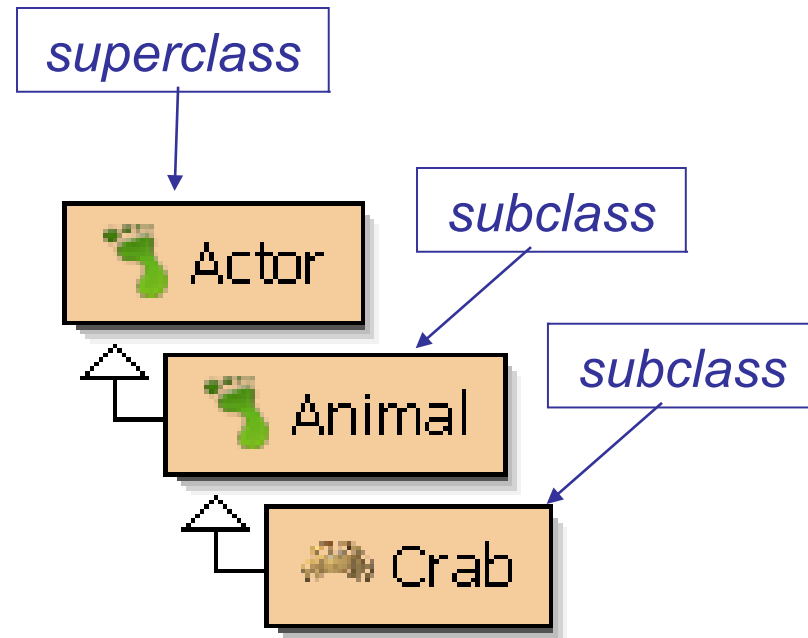
UML for inheritance

Terminology

Different names are used for inheritance relationships.

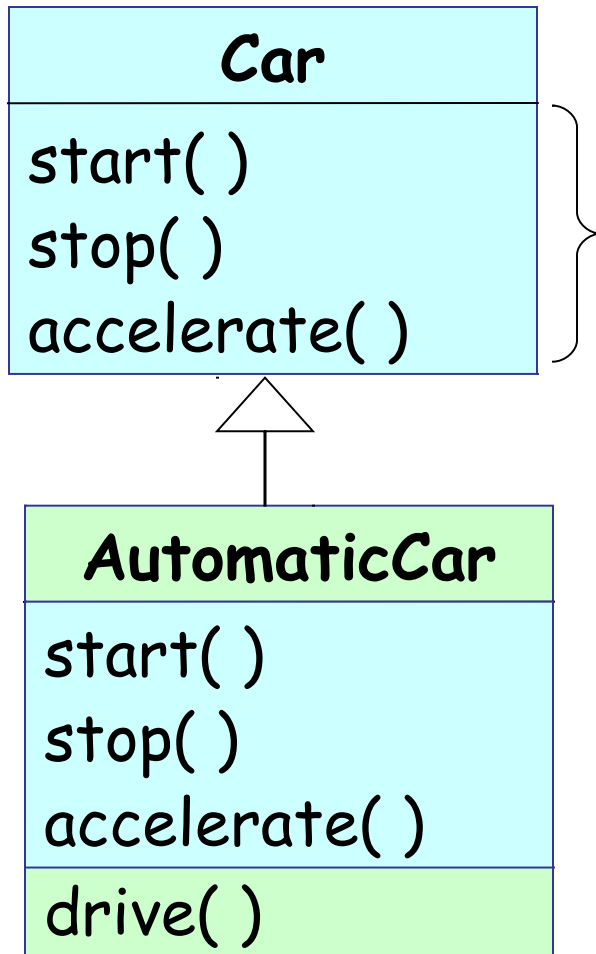
They mean *the same thing*.

Actor	Animal
parent class	child class
superclass	subclass
base class	derived class



Inheritance in Greenfoot.

"Specializing" or "Extending" a Type



Consider a basic **Car**.

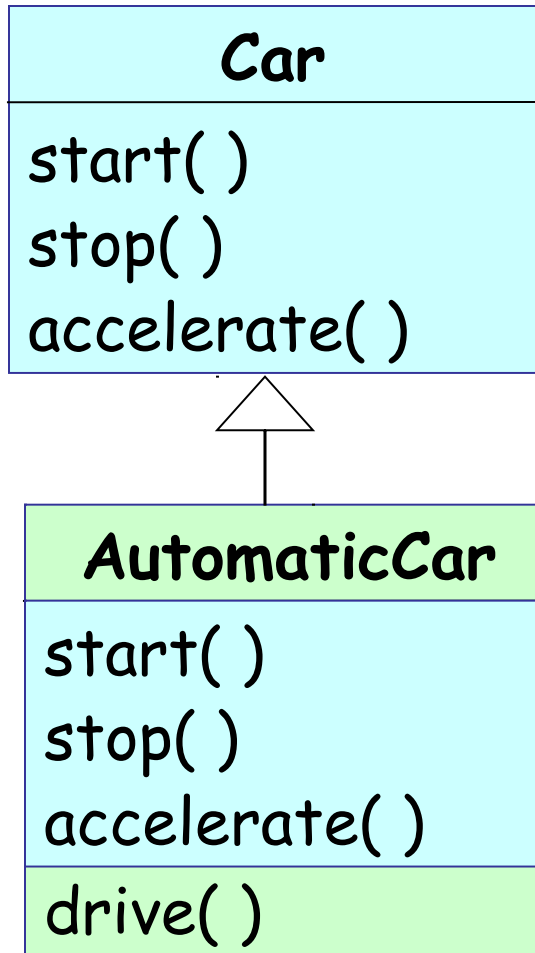
What is the **behavior** of a **Car**?

An **AutomaticCar** is a *special kind* of **Car** with automatic transmission.

AutomaticCar can do **anything** a **Car** can do.

It also adds *extra behavior*.

Benefit of Extending a Type



Extension has some **big benefits**:

Benefit to Car Driver

If you can drive a **basic Car**,
you can drive an **Automatic Car**.
It works (almost) the same.

Benefit to Car Producer

You can **reuse** the **design** and
behavior from **Car** to create
AutomaticCar.
Just add automatic "**drive**".

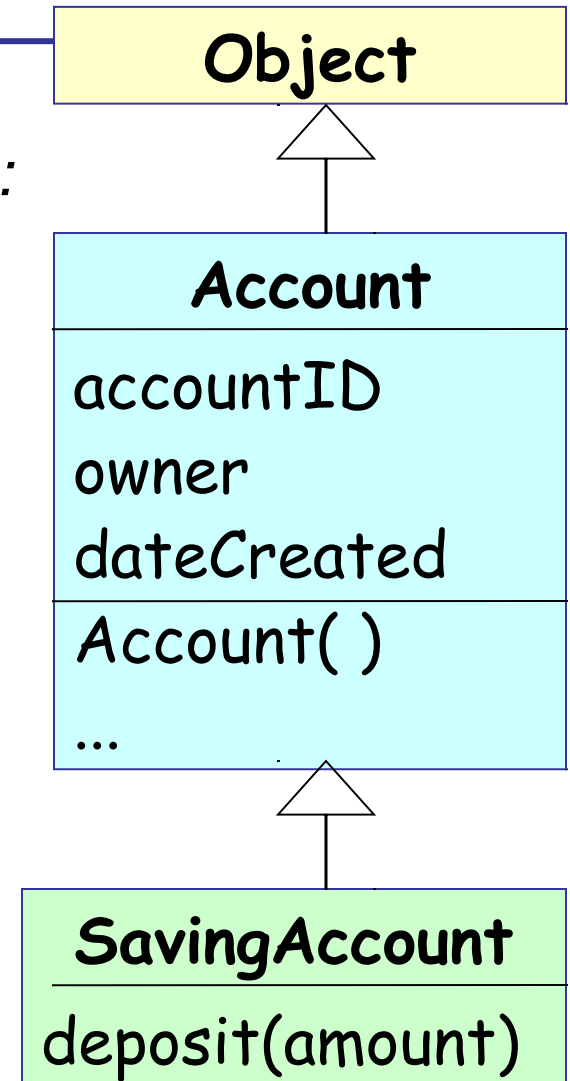
What do you *inherit*?

A subclass *inherits* from its *parent* classes:

- ✓ attributes
- ✓ methods - even private ones.
- ✓ cannot access "**private**" members of parent

In Java, **Object** is a superclass of all classes.

Any method that **Object** has, every class has.



What you don't inherit

A subclass does not inherit

- ✓ constructors
- ✓ static members
- ✓ objects in subclass can invoke them through superclass

Notation for Inheritance

```
class SuperClass {  
    . . .  
}
```



```
class SubClass extends SuperClass {  
    . . .  
}
```

Use "**extends**" and the parent class name.

Interpretation of Inheritance (1)

Superclass defines basic behavior and attributes.

<u>Account</u>
- accountName - accountID # balance
+ deposit(Money) : void + withdraw(Money) : void + toString() : String

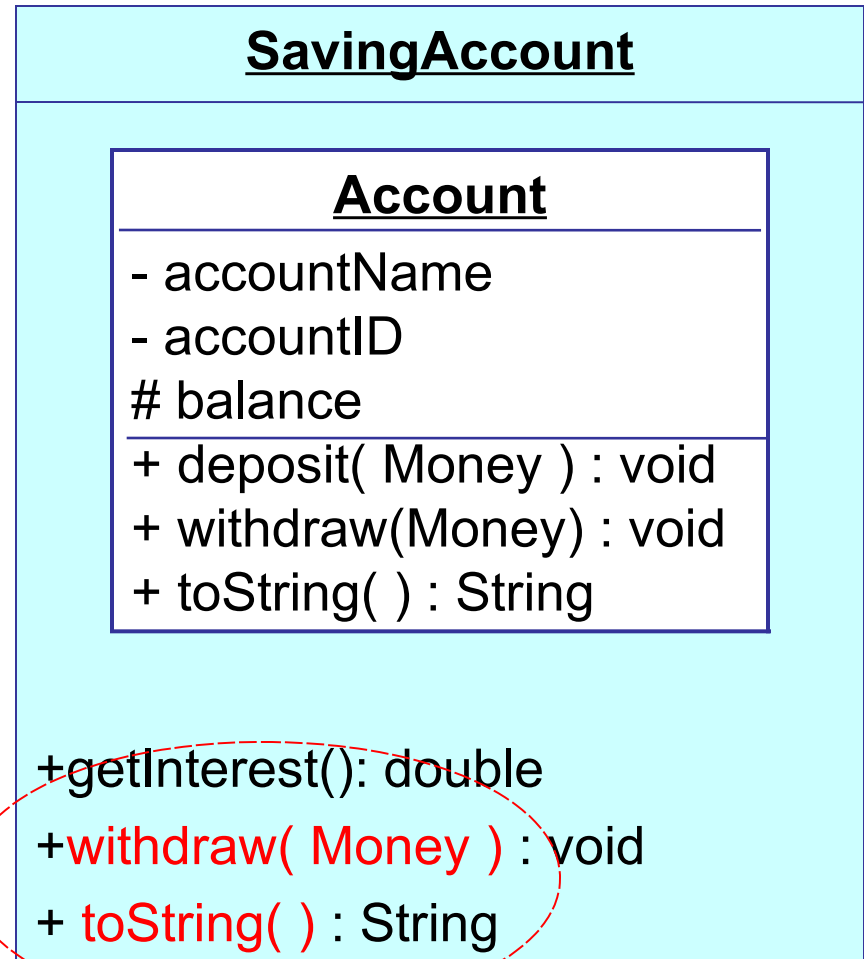
Interpretation of Inheritance (2)

A subclass can...

- ❑ **add new** behavior and attributes (**extension**)
- ❑ **redefine** existing behavior (**specialize**)

Subclass can **override** methods to define its own behavior.

Example: A `SavingAccount` has special rules for withdrawing money.



Using Inheritance: methods

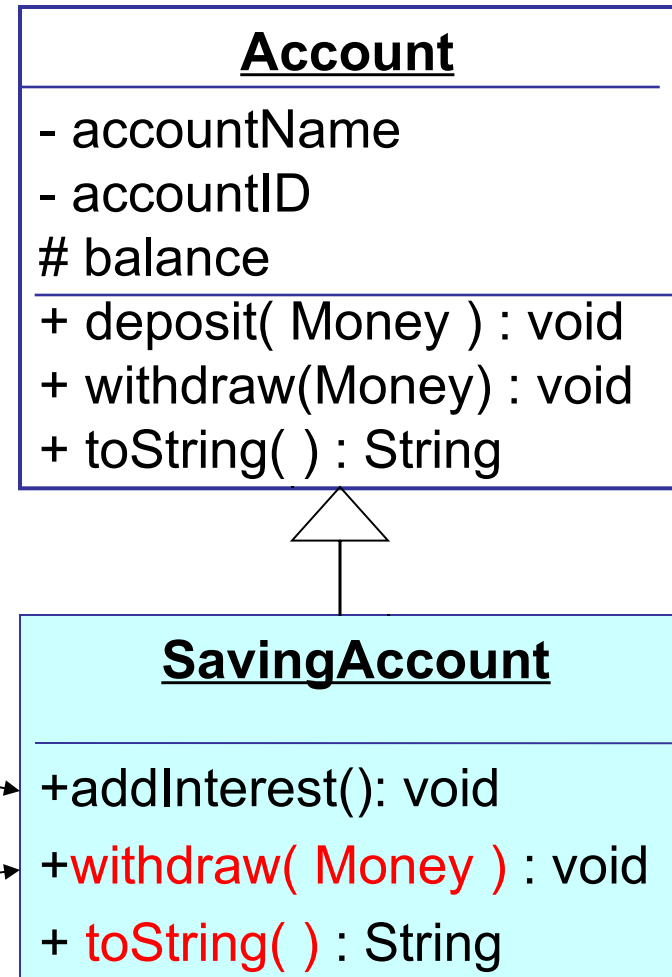
Subclass can:

1) define new methods

2) **override** methods of parent

new method

override



Using Inheritance: attributes

Subclass can access:

- 1) **public** and **protected** attributes of parent
- 2) for **private** attributes must use an *accessor method (provided by the parent class)*

```
class SavingAccount extends Account {  
    public String toString( ) {  
        m = balance;  
        id = getAccountId( );  
    }  
}
```

← **protected** member of Account

← use accessor method to get **private** member of Account

Object: the Universal Superclass

- All Java classes are subclasses of Object.
- You **don't** write "... extends Object".
- Object defines basic methods for all classes:

java.lang.Object

```
#clone()      : Object
+equals(Object) : bool
+finalize()   : void
+getClass()   : Class
+hashCode()   : int
+toString()   : String
+wait()       : void
```

Every class is guaranteed to have these methods.

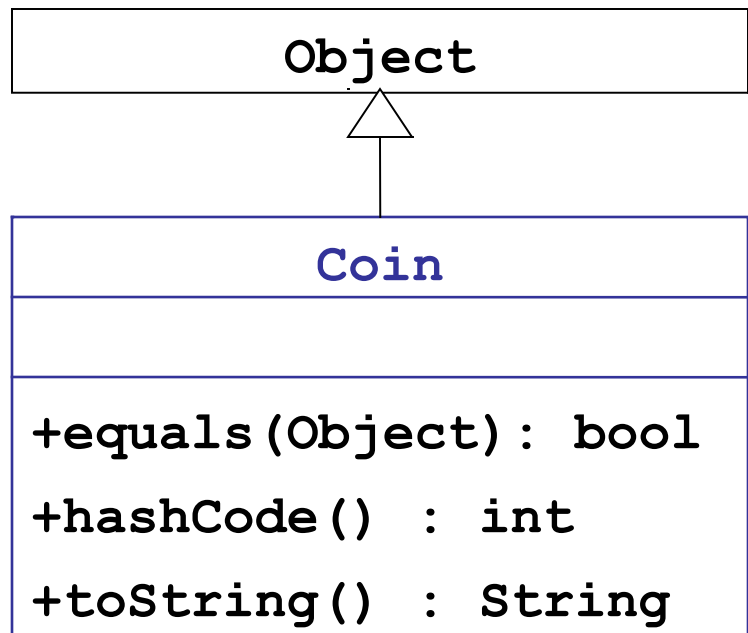
Either:

(1) inherit them

(2) override in subclass

Specializing from Object

- ❑ Most classes want to define their own **equals** and **toString** methods.
- ❑ This lets them *specialize* the behavior for their type.
- ❑ Java automatically calls the class's own method (polymorphism).

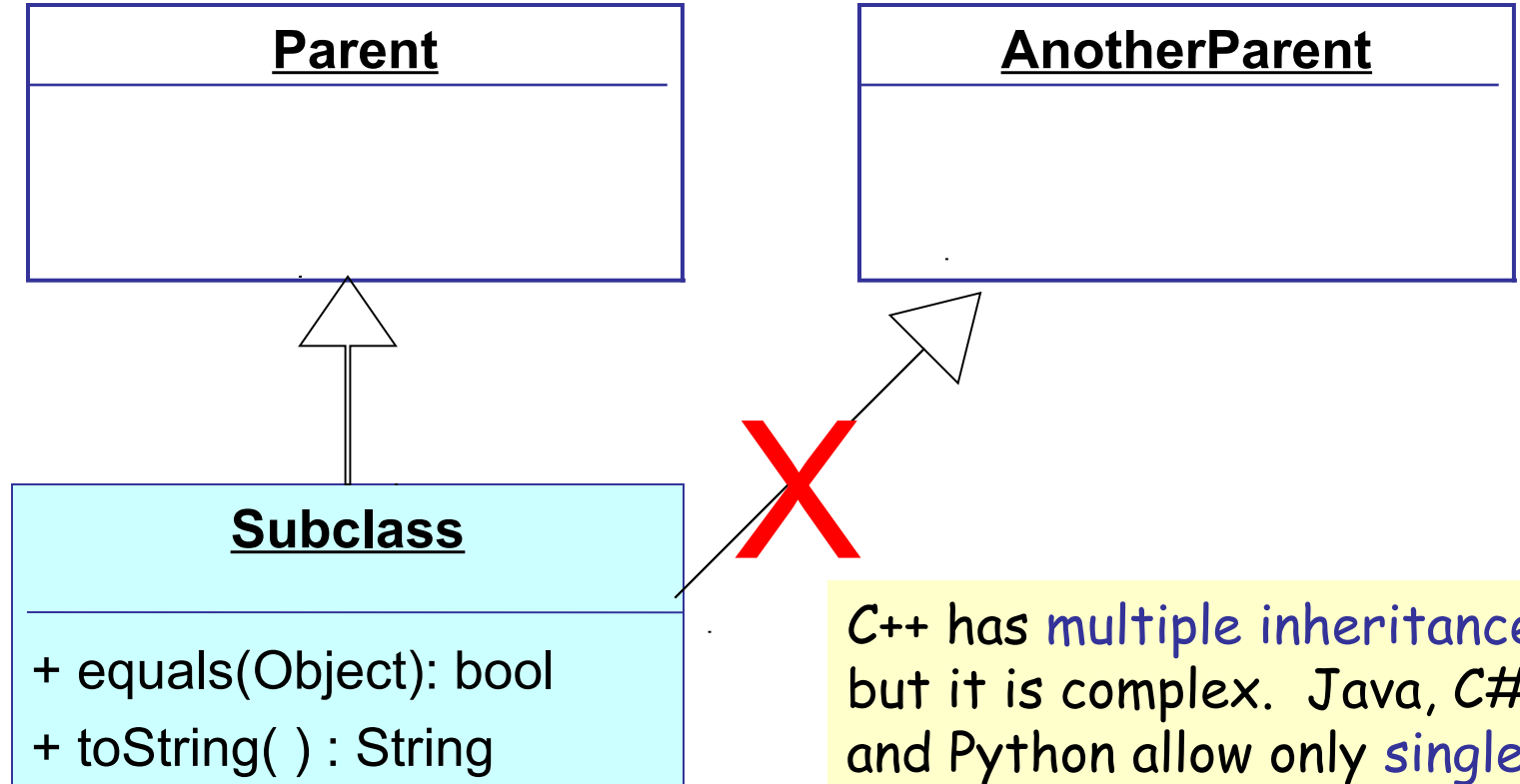


Coin overrides these methods for Coin objects.

A Class has only **One** Parent Class

A class can directly extend **only one** other class.

A class cannot have two parent classes.



C++ has **multiple inheritance**, but it is complex. Java, C#, and Python allow only **single inheritance**.

Why Use Inheritance?

1. *Reuse code.*

subclass can use code from the parent class.

subclass inherits attributes defined in parent class.

2. *Enable polymorphism.*

Object o;

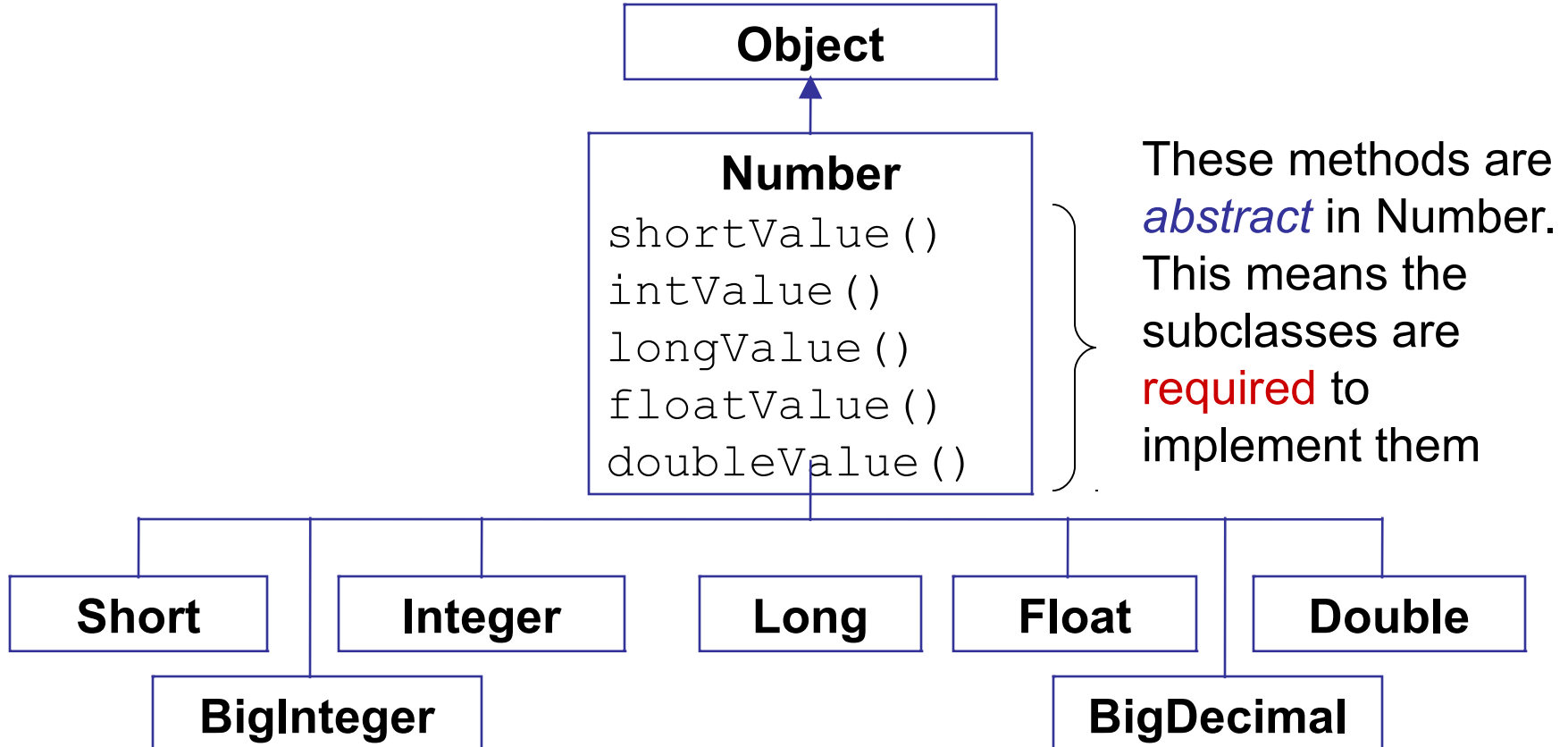
if (Math.random() > 0.5) o = new Date();

else o = new Float(0.25);

System.out.println(o.toString()); // Polymorphism

Number: parent of numeric classes

- Another prodigious parent class is **Number**.
- **Number** defines but does not implement methods that all numeric classes *must* have.



Polymorphism using Number

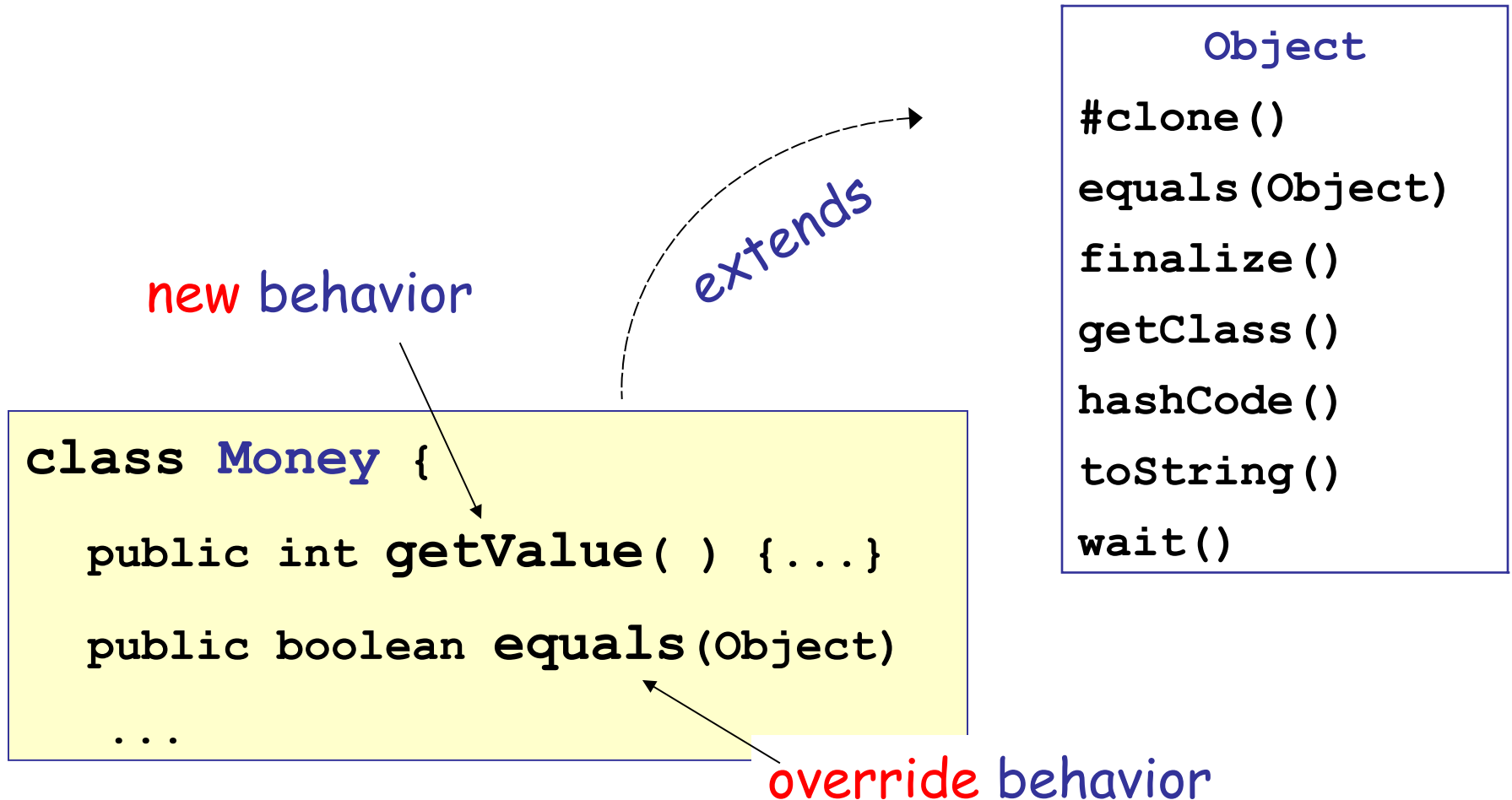
```
public void display(Number num) {  
    System.out.println("The value is "+num.intValue() );  
}  
  
display( new Integer( 10 ) );  
display( new BigDecimal( 3.14159 ) );
```

The value is 10

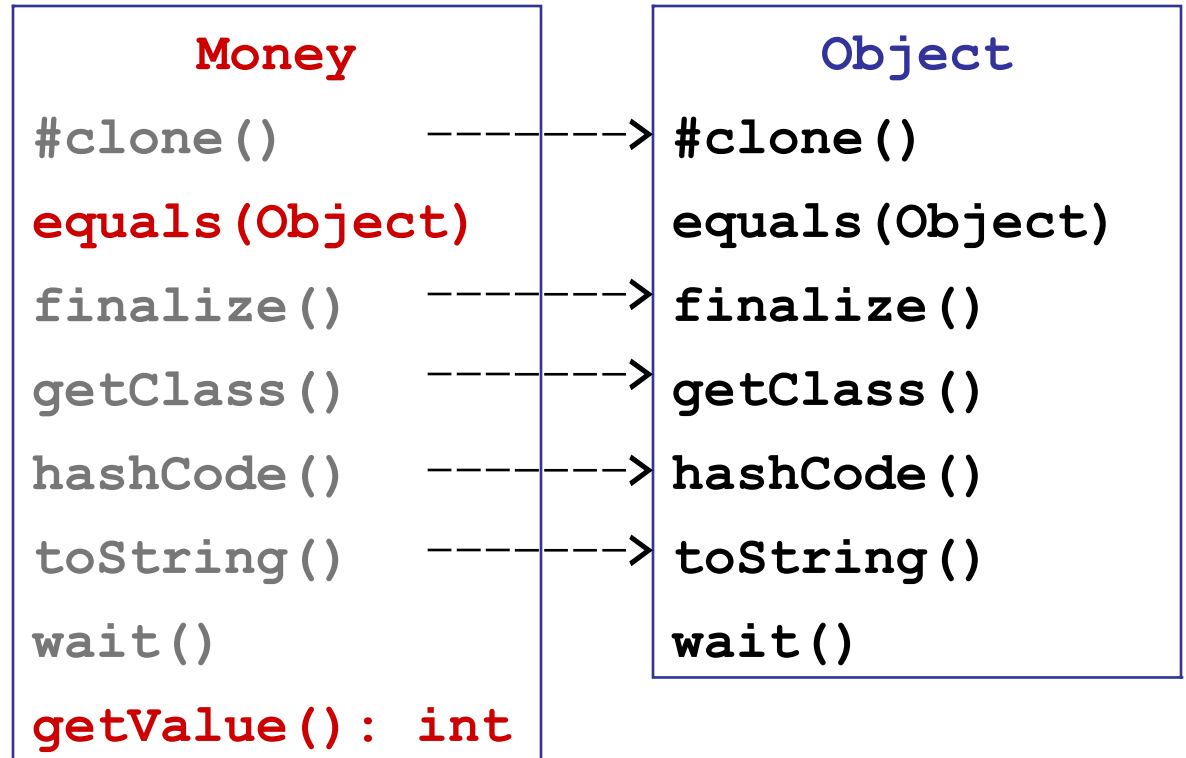
The value is 3

Question: What O-O fundamental enables display to accept a parameter of type Integer or BigDecimal?

Inherited Methods



Inherited Methods



Summary: Override vs New Method

Override method must match the *signature* of the superclass method:

```
public class Money {  
    public int compareTo( Money other )  
}  
  
public class Coin extends Money {  
    public int compareTo( Money other )  
}
```

What Can Override Methods Change

Override method can change **2 things** in signature:

(1) can be **more visible** than parent method

(2) **return type** can be a **subtype** of parent method type

```
public class Purse {  
    protected    List    withdraw( double amount )  
}  
public class MyPurse extends Purse {  
    public        ArrayList withdraw( double amount )  
}
```

The diagram illustrates two Java classes. The first class, `Purse`, has a method `withdraw(double amount)` with the access modifier `protected` and return type `List`. The second class, `MyPurse`, extends `Purse` and overrides the `withdraw` method. In `MyPurse`, the access modifier is `public` (indicated by an upward arrow from `public` to `protected`) and the return type is `ArrayList` (indicated by an upward arrow from `ArrayList` to `List`). This demonstrates that an overriding method can be more visible and have a more specific return type.

New Method, not Override

Any other change defines **a new method**, not an override of parent method.

```
public class Money {  
    public int compareTo( Money other )  
    @Override  
    public boolean equals( Object other )  
}  
  
public class Coin extends Money {  
    public int compareTo( Coin other ) // new method  
    public int compareTo( Coin a, Coin b ) // new method  
    public boolean equals( Coin other ) // new method
```

Why write @Override ?

Enables compiler to detect accidental misspelling, etc.

```
public class Money {  
    @Override // Compile-time ERROR: invalid "override"  
    public boolean equals( Money other ) {  
        return this.value == other.value;  
    }  
  
    // Typing error, but not detected by compiler  
    public String toString( ) {  
        return "Money, money";  
    }  
}
```

Compiler will warn you of misspelled
"toString" if you write @Override

Two uses of @Override

1. In Java 5, @Override meant "override a method"

```
public class Money {  
    @Override  
    public String toString( ) {  
        return "some money";  
    }  
}
```

2. In Java 6+, @Override can also mean "implements"

```
public class Money implements Comparable<Money> {  
    @Override  
    public int compareTo(Money other) {  
        . . .  
    }  
}
```

Cannot Override

- ✓ Constructors

- ✓ static methods
- ✓ private methods



Subclass method creates a new method for these -- having the same name.

- ✓ final methods

Redefining final methods is forbidden. Compile-time error.

final method

```
public class Money {  
  
    public final int getValue( ) { return value; }  
}
```

```
public class Coin extends Money {  
    // ERROR  
    public int getValue( ) { ... }  
}
```

Question: Does Object have any final methods?

Inheritance of Attributes

1. subclass object inherits **all attributes** of the parent class (even the private ones).
 - subclass **cannot directly access private attributes** of the parent -- but they are still part of the object's memory!
2. subclass can **shadow attributes** of the parent by defining a new attribute with the same name.
 - shadow creates a *new attribute having same name as parent's attribute*, but the parent's attributes are still there (just hidden or "shadowed").
 - this is rarely used -- not good design.

Inheritance of Attributes

```
B b1 = new B(12345, "baby" )
```

In memory...

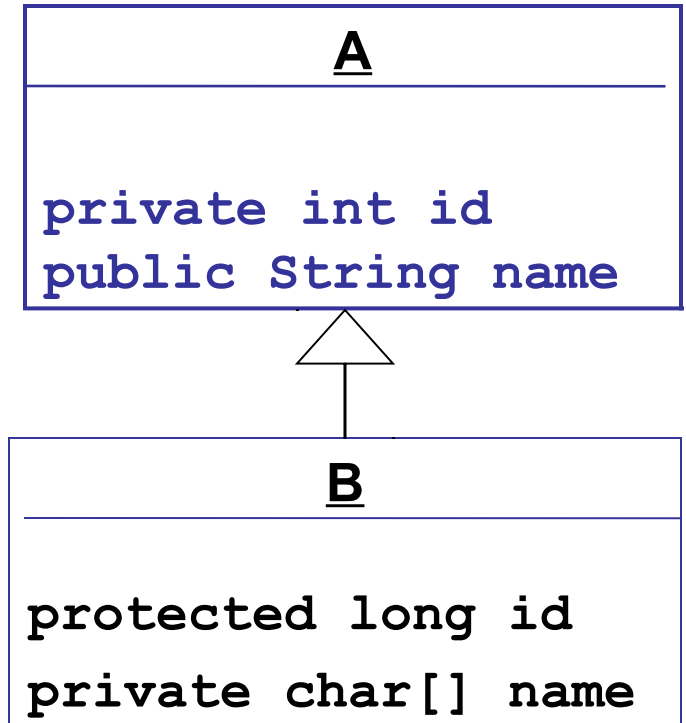
b1: B

long id = 1234567890

char [] name = { 'b','a','b','y' }

(hidden) int id = 0

(hidden) String name = "unknown"



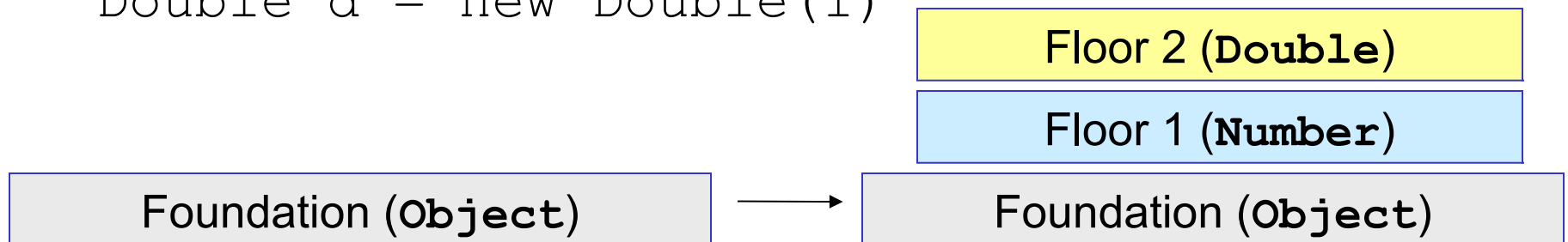
Constructors and Inheritance

To **build** a building...

- first you must build the **foundation**
- then build the **first floor**
- then build the **second floor**
- *etc...* until you build all the floors

Example:

```
Double d = new Double(1)
```



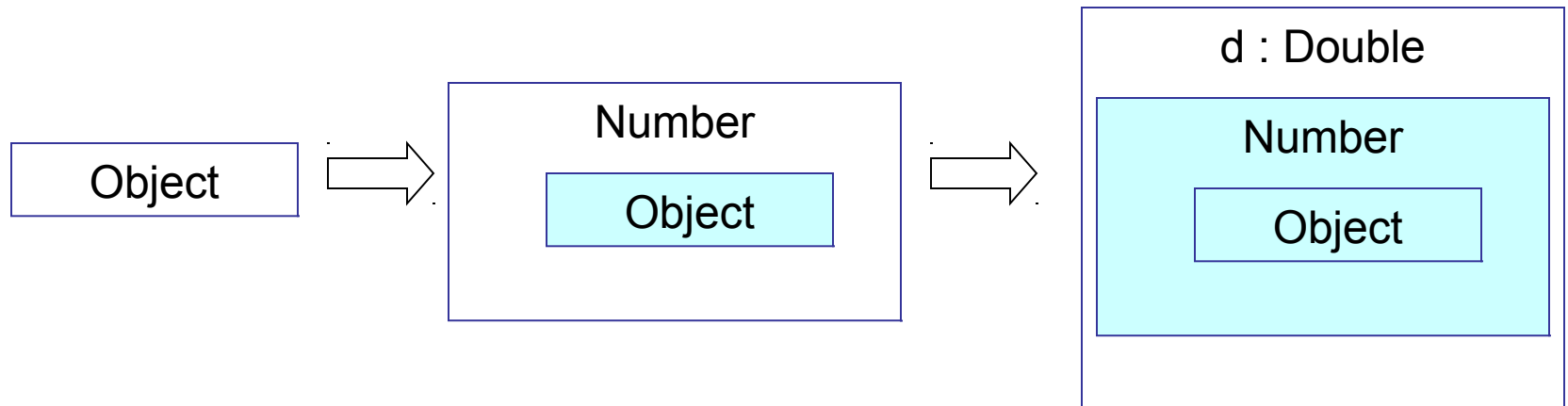
Constructors and Inheritance

To ***construct*** an object of a derived class...

- first you have to build the foundation class (Object)
- then build the 1st subclass of Object
- then build the subclass of the 1st subclass

Example:

```
Double d = new Double( 1.0 );
```

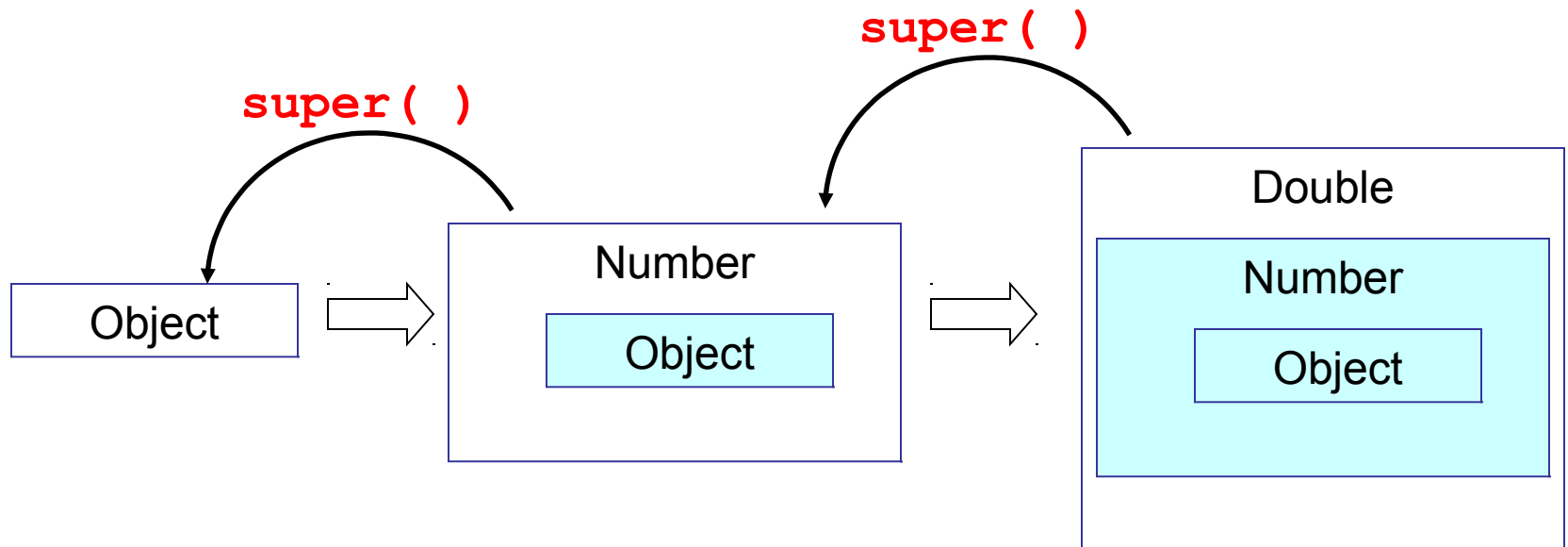


Calling a Superclass Constructor

The subclass constructor must invoke the superclass constructor to "build" the superclass object.

2 ways:

- *explicitly* call a superclass constructor in code
- *implicitly* (by Java compiler) invoke the superclass *default* constructor

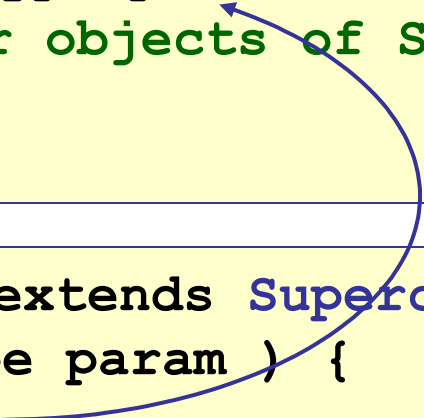


Explicitly Call Superclass Constructor

- ❑ A subclass can call a superclass constructor using the reserved name: **super**(...)
- ❑ **super** must be the **first statement** in the constructor.

```
public class Superclass {  
    public Superclass( Type param ) {  
        // constructor for objects of Superclass  
    }  
}
```

```
public class Subclass extends Superclass {  
    public Subclass( Type param ) {  
        super( param );  
    }  
}
```



Implicit Call to Superclass Constructor

- If a class does not explicitly call a "super" constructor, then Java will automatically insert a call to **super ()**
- Java **always** calls the superclass **default constructor**

```
public class Object {  
    public Object( ) { /* constructor for Object class */ }  
}
```

```
public class Superclass extends Object {  
    public Superclass( ) { /* default constructor */ }  
}
```

super ()



```
public class Subclass extends Superclass {  
    public Subclass( Type1 arg1, Type2 arg2 )  
    { this.property1 = arg1;  
      this.property2 = arg2;  
    }  
}
```

super ()



Error in Implicit super() Call

- If SavingAccount does not explicitly call "super (name)". What happens?

```
public class SavingAccount extends BankAccount {  
    public SavingAccount(String name) {  
        <-- implicit call to super( )  
        if (Bank.creditCheck(name) == Bank.BADCREDIT)  
            overdraftLimit = 0;  
        ...  
    }  
}
```

The Java compiler issues an error message:

```
Implicit super constructor BankAccount( ) is  
undefined.
```

Why the Error?

- ❑ **BankAccount** doesn't have a **default** constructor, so we get an error.
- ❑ This error is good!

It reminds us that we must call the right constructor of **BankAccount**.
- ❑ For hierarchies such as **Number**, the parent class should define a default constructor to avoid errors in subclasses.



Inheritance and Polymorphism

How inheritance and run-time "binding" of
method names to method code enable
polymorphism

Binding of Methods to References

- Java determines which *instance method* should be called for a method name at run-time.
- This is called **dynamic binding** or **late binding**.
- This means that you can't tell which **actual method** will be called from only the variable type.

```
Object obj = "What am I?"; // obj -> String
if (Math.random() > 0.5)
    obj = new Date( );

// which toString will be used?
obj.toString( );
```

Binding a method name to code

Compile Time Binding

Compiler "binds" a method name to code using the **declared class** of the **variable**

- ❑ most efficient
- ❑ no polymorphism

When is this used?

- ❑ "final" methods
- ❑ "final" class
- ❑ private methods
- ❑ static methods
- ❑ constructors
- ❑ "value" types (**C#**: **struct**)

Runtime Binding

Method is invoked using the **actual type** of the **object**.

- ❑ slower
- ❑ enables polymorphism

When is this used?

- ❑ **Java**: all methods except "final", "static", or "private"
- ❑ **C#**: only for **virtual** methods

Overriding Methods and access

Q: Can a subclass change the visibility of a method that it overrides?

A: a subclass can *increase the visibility* of method it overrides, but it cannot *decrease* the visibility.

Method in Superclass

`public`

`protected`

`package (default)`

`private`

Method in Subclass

`public`

`public protected`

`public protected package`

`anything`

Overriding Methods (1): visibility

```
class BankAccount {
    public boolean withdraw( double amount ) {
        ....
    }
}

class CheckingAccount extends BankAccount {
    ??? boolean withdraw( double amount ) {
        ....
    }
}
```

The Test: does polymorphism work?

```
BankAccount b = new BankAccount( "Mine" );
BankAccount c = new CheckingAccount( "Yours" );
b.withdraw( 100 ); // if this is OK
c.withdraw( 100 ); // then will this be OK?
```

Overriding Methods (2): visibility

Q: Can a subclass change the visibility (access privilege) of a method that it overrides?

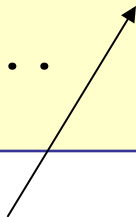
- ❑ change access from "public" to "protected":

```
class CheckingAccount extends BankAccount {  
    protected void withdraw( double amount ) {  
        if ( amount > balance + overDraftLimit ) {  
            System.out.printf(  
                "Error: you can withdraw at most %f Baht\n",  
                balance+overDraftLimit );  
            return /*false*/; // cannot withdraw  
        }  
    }  
}
```

This method is "**public**" in the BankAccount class.

Overriding Methods (3): return type

```
class BankAccount {  
    public boolean withdraw( double amount ) {  
        ....  
    }  
}  
class CheckingAccount extends BankAccount {  
    public void withdraw( double amount ) {  
        ....  
    }  
}
```



Can a subclass change the **return type** of overridden method?

The Test: does polymorphism work?

Overriding Methods (4): parameters

Q: Can a subclass change the type of a parameter of an overridden method?

Example: change **amount** from "double" to "long":

```
class BankAccount {  
    public boolean withdraw( double amount ) { ... }  
    ....  
}  
  
class CheckingAccount extends BankAccount { /**  
    withdraw method for checking account */  
    public boolean withdraw( long amount ) { ... }  
}
```

Overriding Methods: parameters

Answer: Yes, but then you aren't overriding the method!

If the parameter type is different then you are creating a new method with the same name (called "method overloading").

```
/** test the withdraw method */
public void testWithdraw( ) {
    CheckingAccount ca = new CheckingAccount("...");
    ca.withdraw( 50000 );
    // this calls CheckingAccount.withdraw()
    ca.withdraw( 25000.0 );
    // calls BankAccount.withdraw()
```

Overriding Methods (5): using `super`

Q: Can we access the method of the superclass, even though it has been overridden?

- invoke `withdraw` of `BankAccount` using `"super"`.

```
class CheckingAccount extends BankAccount {  
    public boolean withdraw( long amount ) {  
        if ( overDraftLimit == 0 )  
            super.withdraw(amount); // parent's method  
        else if ( amount > balance + overDraftLimit )  
            System.out.printf("Error: ...");  
        else  
            balance = balance - amount;  
    }  
}
```

Overriding Methods (6): using **super**

Consider a **Person** superclass and **Student** subclass.

- ❑ (Person) p.compareTo() compares people by name
- ❑ (Student) s.compareTo() compares by student ID first and then name.

```
public class Student extends Person {  
    private String studentID;  
    public int compareTo(Object other) {  
        ... // test for null and Student type  
        Student s = (Student) other;  
        int comp = studentID.compareTo(s.studentID);  
        if ( comp != 0 ) return comp;  
        // if studentID is same, compare by name  
        return super.compareTo(other) ;  
    }  
}
```

Redefining Attributes

A subclass can declare an attribute with the same name as in the superclass.

The subclass attribute *hides* the attribute from parent class, but it still inherits it!

You can see this in BlueJ by "inspecting" an object.

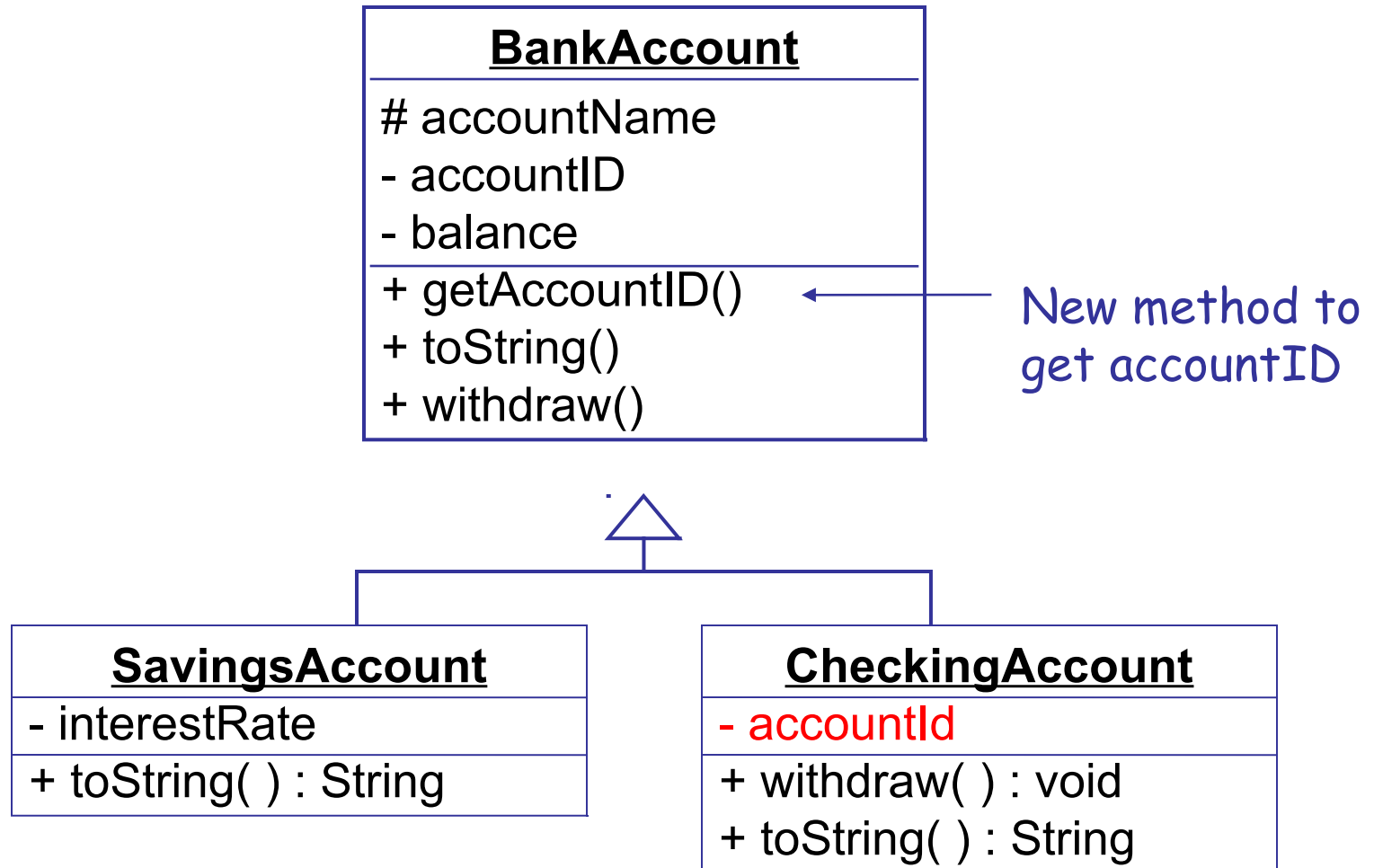
```
public class BankAccount {  
    private long accountId;  
}
```

```
public class SavingAccount  
    extends BankAccount {  
    private String accountId;  
}
```

SavingAccount has 2 id attributes. The parent attribute is private (not accessible) and hidden by its own attribute.

Redefining Attributes

The new BankAccount hierarchy is:



Questions About Redefined Attributes

If a subclass redefines an attribute of the superclass...

- ❑ can the subclass still access the superclass attribute?
- ❑ can the subclass change the visibility of the attribute?

Example: in CheckingAccount can we declare:

```
private long accountID;
```

- ❑ can the subclass change the datatype of the attribute?

Example: in CheckingAccount can we declare:

```
protected String accountID;
```

Review Questions

Object References

Q1: Which of these assignments is legal?

```
/* 1 */  
BankAccount b = new CheckingAccount("Nok");  
/* 2 */  
CheckingAccount c = new BankAccount("Noi");  
/* 3 */  
Object o = new BankAccount("Maew");  
/* 4 */  
BankAccount b = new Object( );
```

Object References

Q2: What is the effect of this reassignment?

```
BankAccount ba;  
CheckingAccount ca = new CheckingAccount("Noi");  
ca.deposit( 100000 );  
// assign to a BankAccount object  
ba = ca;
```

What happens when "**ba = ca**" is executed?

1. It converts CheckingAccount object to a BankAccount object. Any extra attributes of CheckingAccount are lost!
2. It converts CheckingAccount object to a BankAccount object. Any extra attributes of CheckingAccount are hidden until it is cast back to a CheckingAccount object.
3. Has no effect on the object.
4. This statement is illegal.

I Want My Checking Account!

Q3: Suppose a `BankAccount` *reference* refers to a `CheckingAccount` *object*. How can you assign it to a `CheckingAccount`?

```
BankAccount ba = new CheckingAccount("Jim");
CheckingAccount ca;
if ( ba instanceof CheckingAccount ) {
    // this is a checking account.
    ca = ??? ; // make it look like a checking acct
    how can you assign the bank account(ba) to ca ?
}
```

1. `ca = ba;`
2. `ca = new CheckingAccount(ba);`
3. `ca = ba.clone();`
4. `ca = (CheckingAccount) ba;`
5. none of the above.

Overriding equals()

- ❑ The Object class contains a public `equals()` method.

Q1: Does BankAccount `equals()` override the Object `equals()` method?

```
/** compare two BankAccounts using ID */  
public boolean equals( BankAccount other ) {  
    if ( other == null ) return false;  
    return accountID == other.accountID;  
}
```

```
Object a = new Object( );  
BankAccount b = new BankAccount( "Joe" );  
if ( b.equals( a ) )  
    System.out.println("Same");
```

Overriding equals()

- The Object class contains a public `equals()` method.

Q2: CheckingAccount does not have an equals method.
Which equals will be called here?

```
/** compare two Checking Accounts */
CheckingAccount ca1 = new CheckingAccount(...);
CheckingAccount ca2 = new CheckingAccount(...);
...
if ( ca1.equals(ca2) ) /* accounts are same */
```

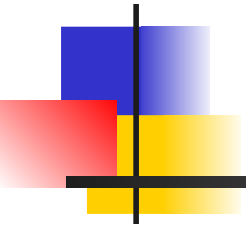
1. `(BankAccount)equals`
2. `(Object)equals`
3. neither. Its an error because CheckingAccount doesn't have equals.

Homework: Binding of Methods

Homework

There are at least 3 situations where Java "binds" a method name to an actual method at **compile time** (more efficient for execution).

- > What are these situations?
- > Give an example of each.



Summary of Important Concepts

Subclass has all behavior of the parent

- A subclass inherits the attributes of the superclass.
- A subclass inherits behavior of the superclass.
- Example:

Number has a **longValue()** method.

Double and **Fraction** are subclasses of **Number** .

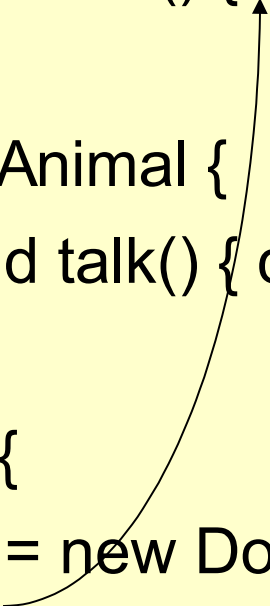
Therefore, **Double** and **Fraction** also have **longValue()**

Java

```
class Animal {  
    void talk() { console.print("grrrrr"); }  
}  
class Dog extends Animal {  
    void talk() { console.print("woof"); }  
}  
void main() {  
    Animal a = new Dog();  
    a.talk( ); <--- which talk method is invoked?  
}
```

C#

```
class Animal {  
    public void talk() { console.write("grrrrr"); }  
}  
class Dog : Animal {  
    public void talk() { console.write("woof"); }  
}  
void main() {  
    Animal a = new Dog();  
    a.talk( ); <--- which talk method is invoked?  
}
```



Polymorphism in C#

```
class Animal {  
    virtual void talk() { console.write("grrrrr"); }  
}  
class Dog : Animal {  
    override void talk() { console.write("woof"); }  
}  
void main() {  
    Animal a = new Dog();  
    a.talk( ); <--- which talk method is invoked?  
}
```

Preventing Inheritance: *final class*

A "final" class cannot have any subclasses.

All "enum" types are final.

String, Double, Float, ... classes are final.

```
public final class String {  
    ...  
}
```

Prevent Overriding: *final* methods

- A "**final**" method cannot be overridden by a subclass.
- **final** is used for important logic that should not be changed.

```
public class BankAccount {  
    // don't let subclasses change deposit method  
    public final void deposit(Money amount) {  
        ...  
    }  
}
```