



Introduction to Inheritance

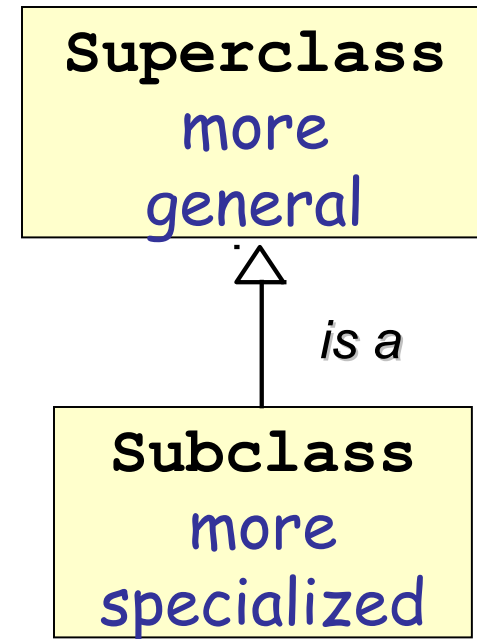
These slides cover the basics of inheritance.

For deeper understanding, see the textbook and 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.
- ❑ Subclass can *redefine* some inherited behavior, or *add new* attributes and behavior.



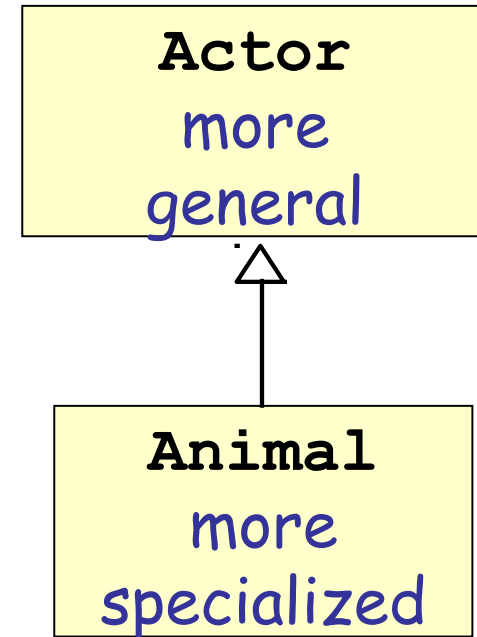
UML for inheritance

Terminology

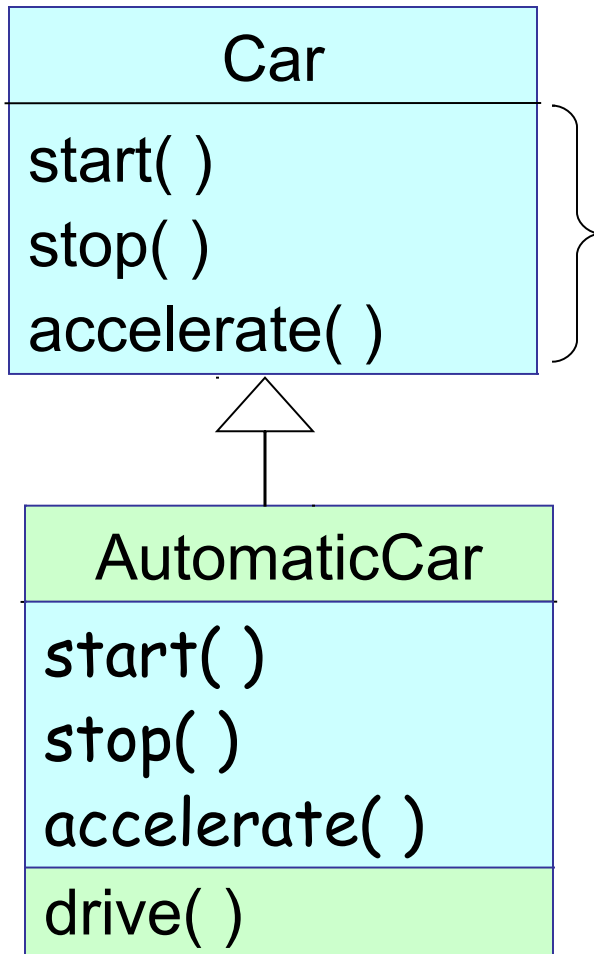
Different names are used for inheritance relationships.

They mean *the same thing*.

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



"Specializing" or "Extending" a Type



Consider a basic **Car**.

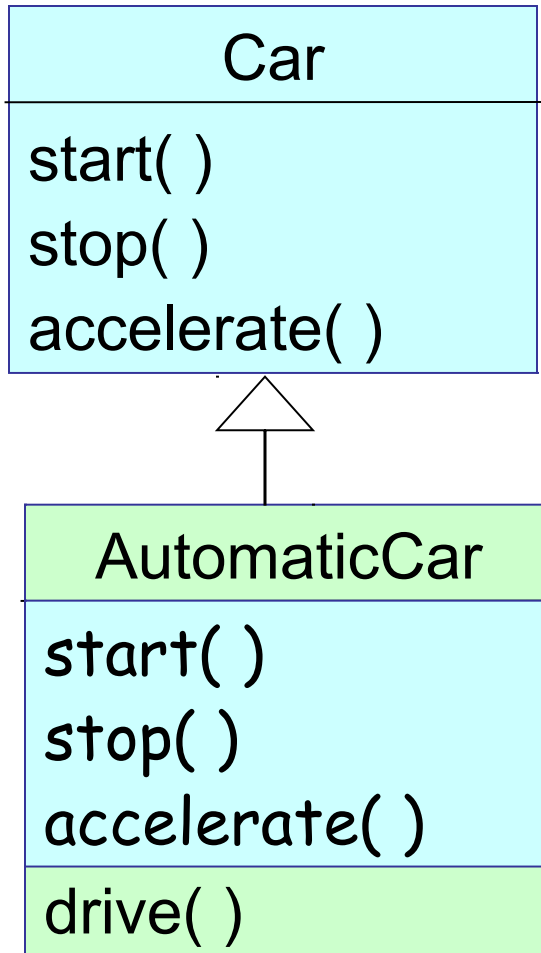
What is the **behavior** of a **Car**?

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 **benefits**:

Benefit to user

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

Benefit to producer (programmer)

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

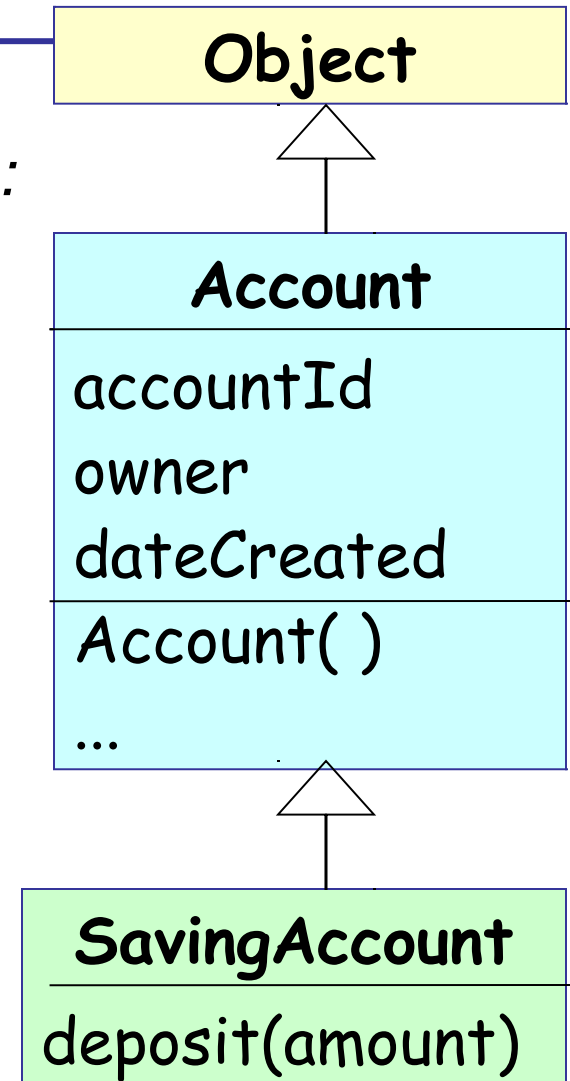
What do you *inherit*?

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

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

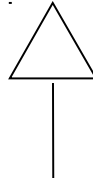
In Java, **Object** is a superclass of all classes.

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



Java Syntax 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

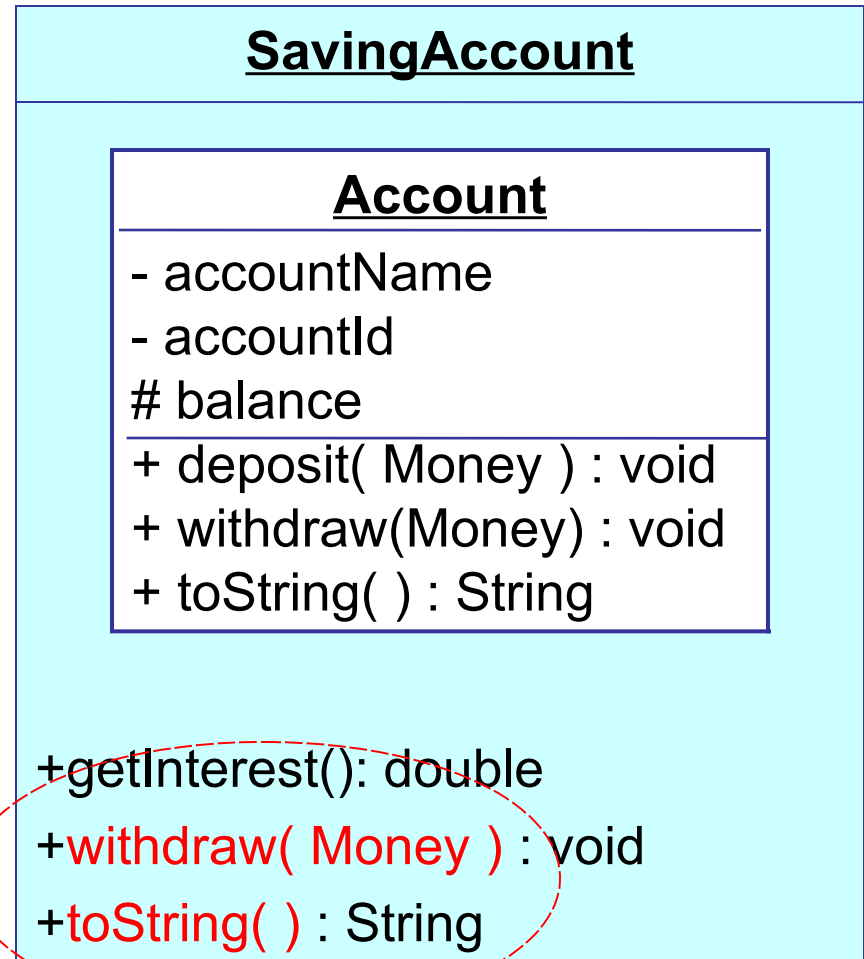
Interpretation of Inheritance (2)

A subclass can...

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

Subclass can **override** methods to specialize its behavior.

SavingAccount **overrides** withdraw and toString.



Attributes and Inheritance

Subclass can access:

- 1) **public** and **protected** attributes of parent
- 2) cannot access **private** attributes. Must use an *accessor method (of the parent class)*

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

← **protected** member of Account

← **private** accountId of Account -
use accessor ("get") method.

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.

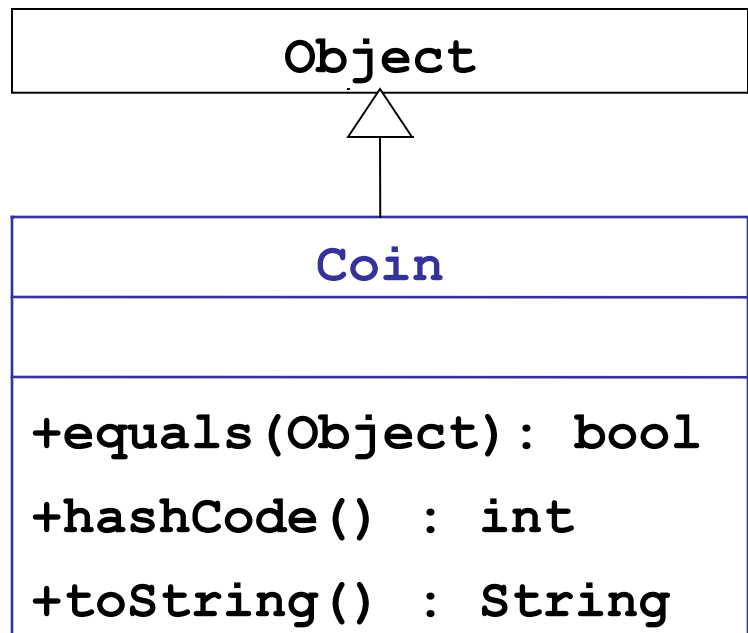
Either:

(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).

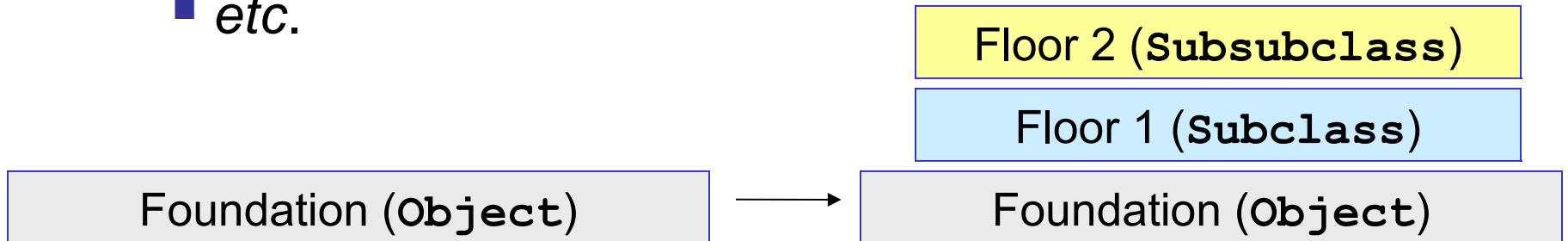


Coin overrides these methods for Coin objects.

Constructors and Inheritance

To **build** a building...

- first you build the **foundation**
- then build the **first floor**
- then build the **second floor**
- *etc.*

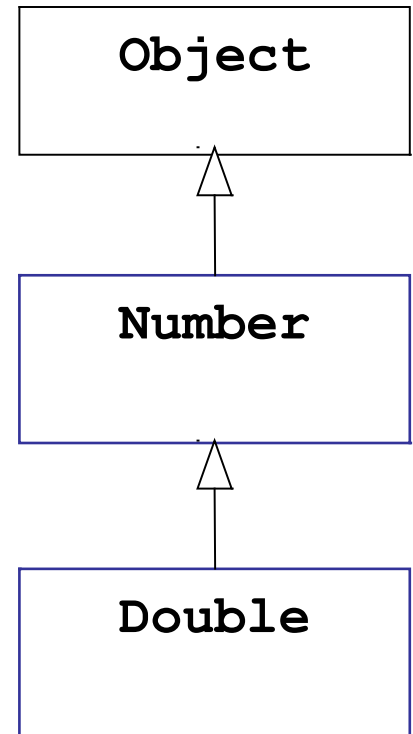


Constructors and Inheritance

Example: Double is subclass of Number

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

What happens?

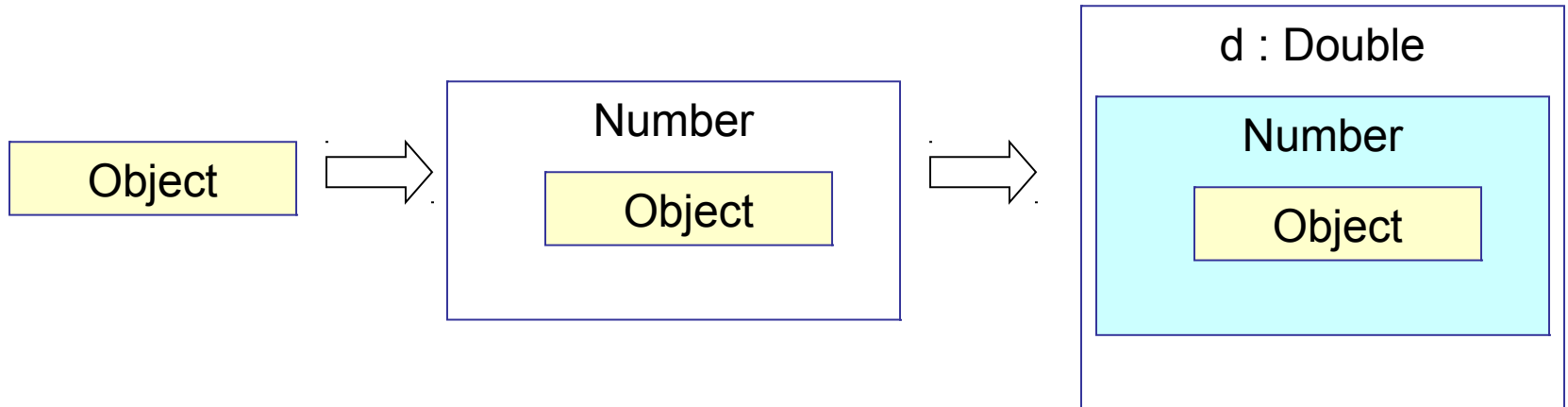


Constructors and Inheritance

Building a Double object:

- initialize the foundation object (**Object**)
- initialize the 1st subclass object (**Number**)
- initialize the 2nd subclass object (**Double**)

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



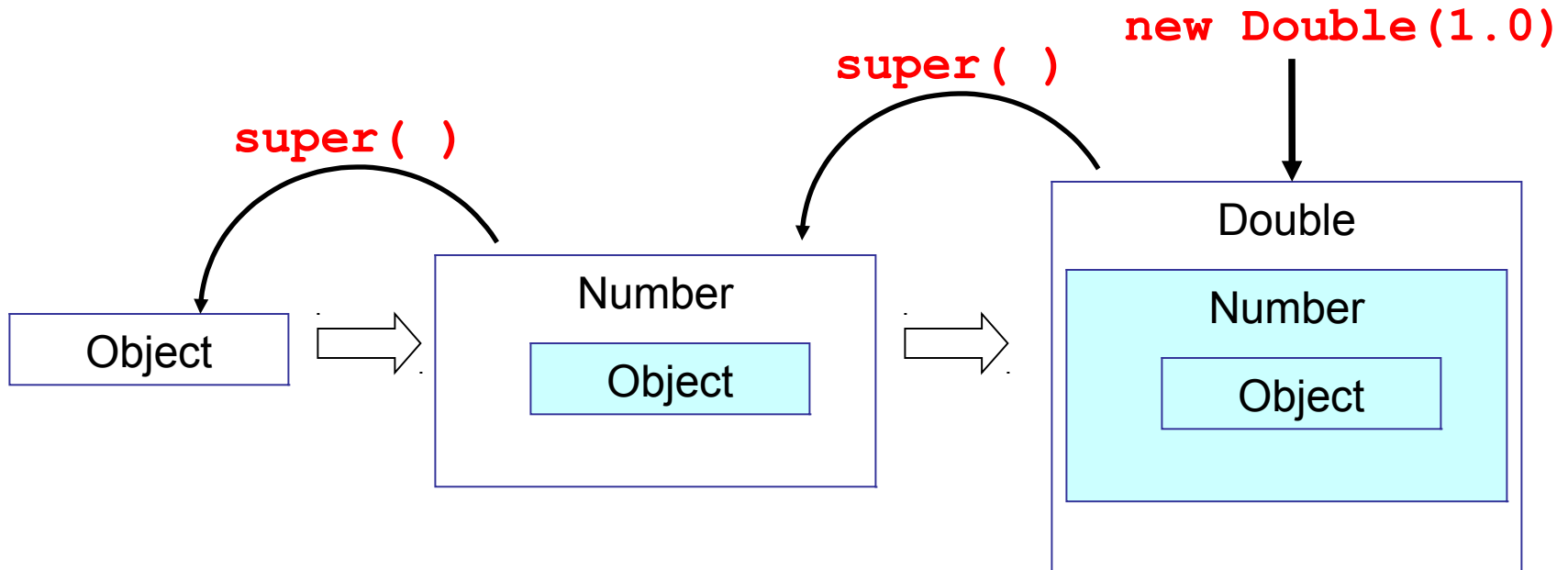
Calling a Superclass Constructor

When you invoke an object's constructor, it *always* calls a constructor of its superclass.

Example:

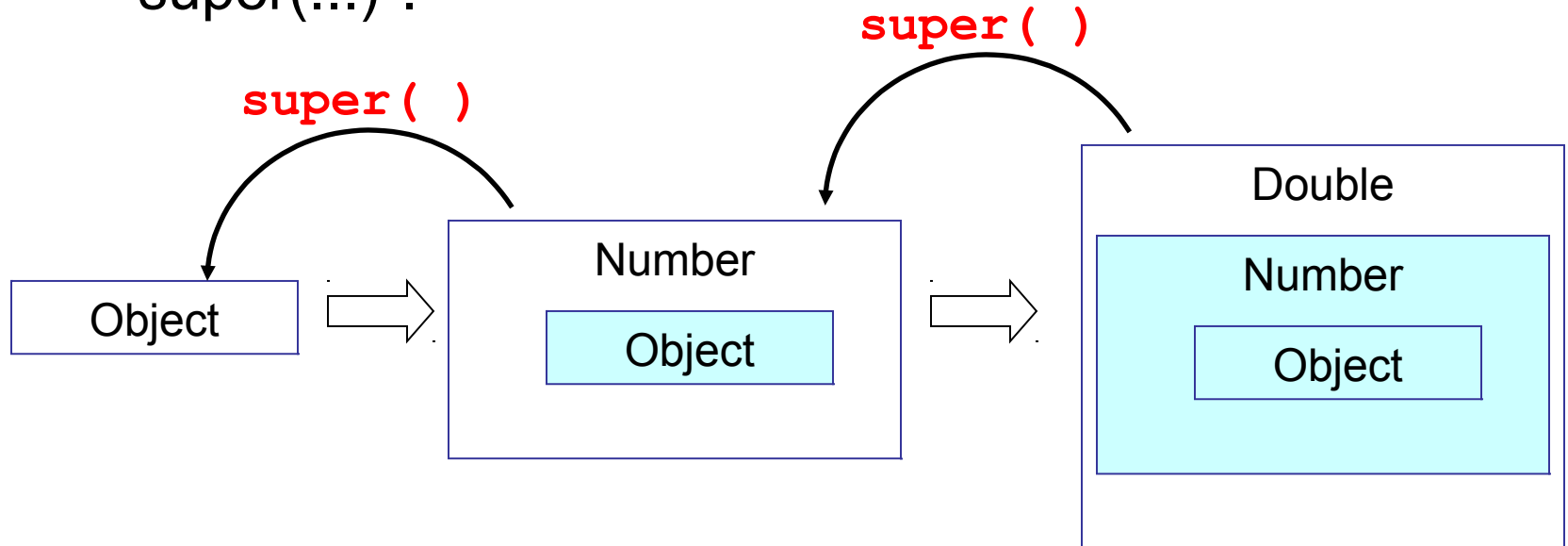
```
Double d = new Double(1.0);
```

- *implicitly* calls `Number()`, which calls `Object()`.



2 Ways to Call Superclass Constructor

- *explicitly* write `super()` to invoke super constructor
- *implicitly* invoke the superclass default constructor. Java compiler does this if you don't explicitly call "`super(...)`".

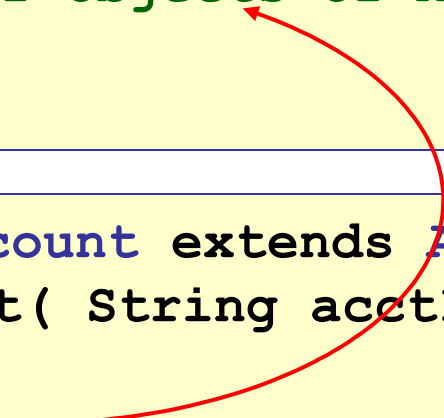


Explicitly Call Superclass Constructor

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

```
public class Account {
    public Account( String acctId ) {
        // constructor for objects of Account class
    }
}
```

```
public class SavingAccount extends Account {
    public SavingAccount( String acctId, String name)
    {
        super( acctId );
    }
}
```



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 calls the superclass default constructor

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

```
public class Number extends Object {  
    public Number( ) { // default constructor  
        super ( )  
    }  
}
```

```
public class Double extends Number {  
    public Double( double value )  
    {  
        super ( )  
        this.value = value ;  
    }  
}
```

Error in automatic call to super()

- If superclass does not have a default constructor, you will get an error from compiler. In SavingAccount:

```
public class SavingAccount extends Account {
    public SavingAccount(String acctId, String name)
    {
        implicit call to super( )

        // initialize SavingAccount attributes
        this.name = name;
    }
}
```

The Java compiler issues an error message:

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

Why the Error?

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

It tells us that we must invoke the right constructor of **Account**.

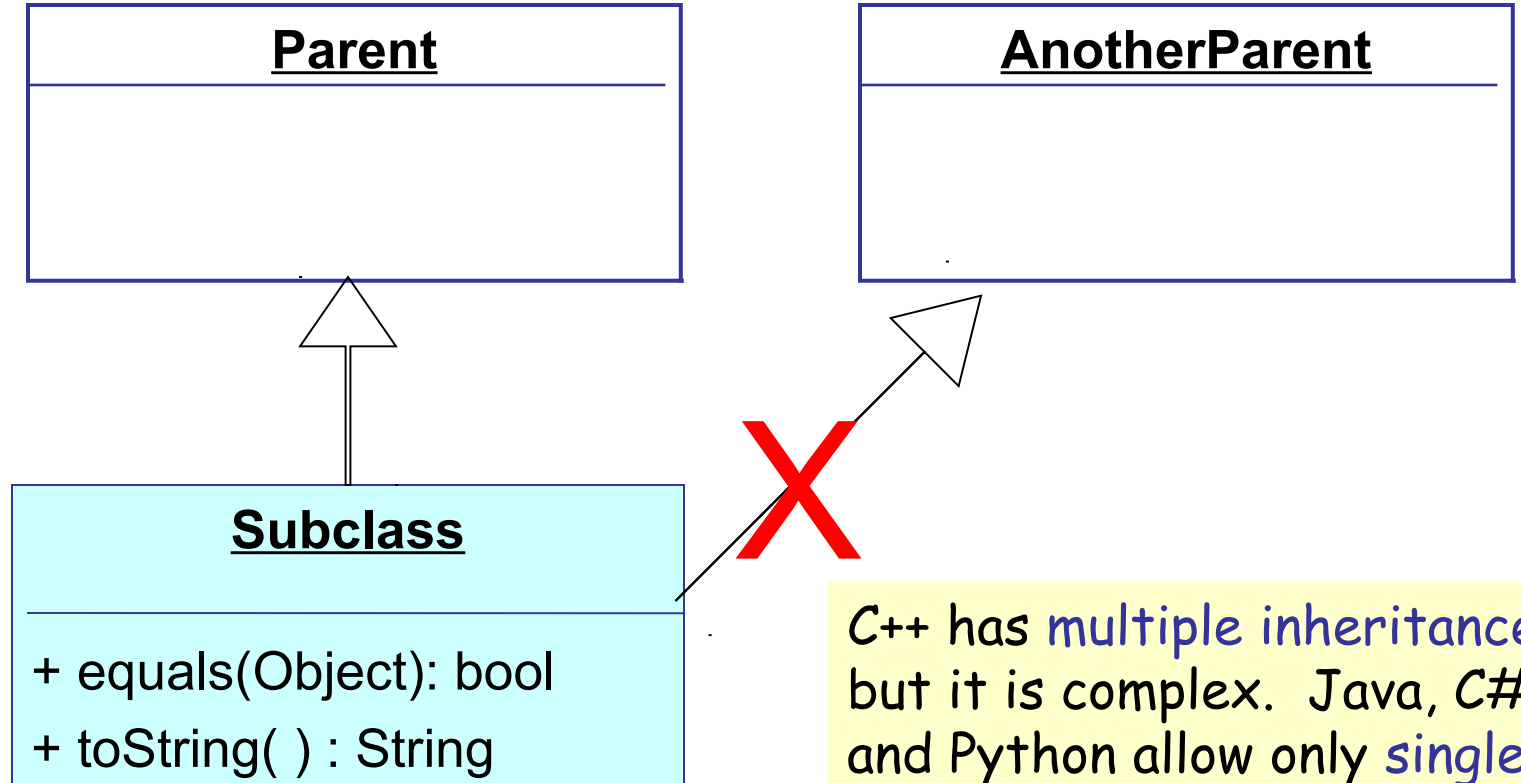
Lesson:

- If superclass does not have a default constructor, then subclasses **must explicitly** write: `super(arguments)`

A Class has only **One** Parent Class

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

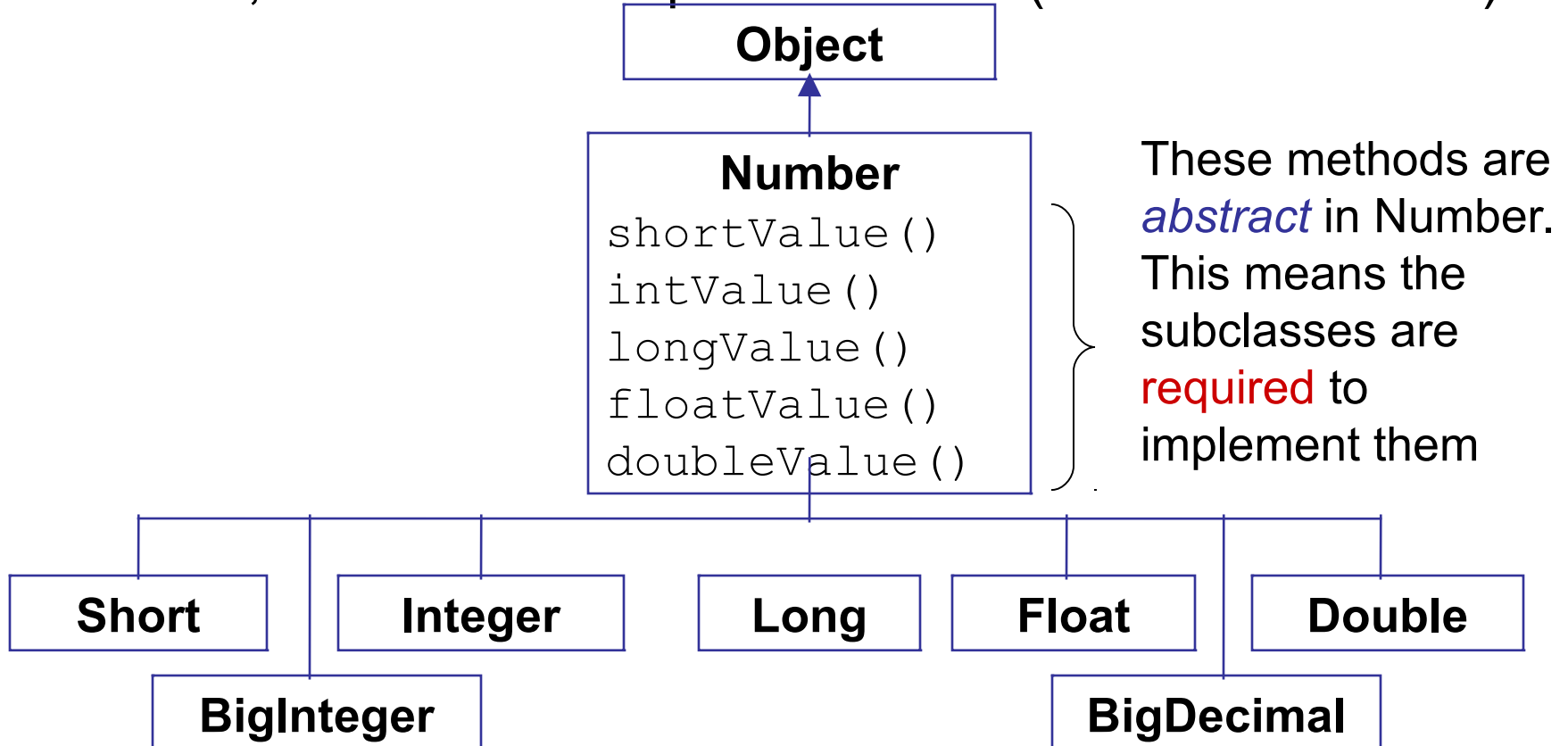
A class cannot have two parent classes.



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

Number: parent of numeric classes

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



Polymorphism using Number

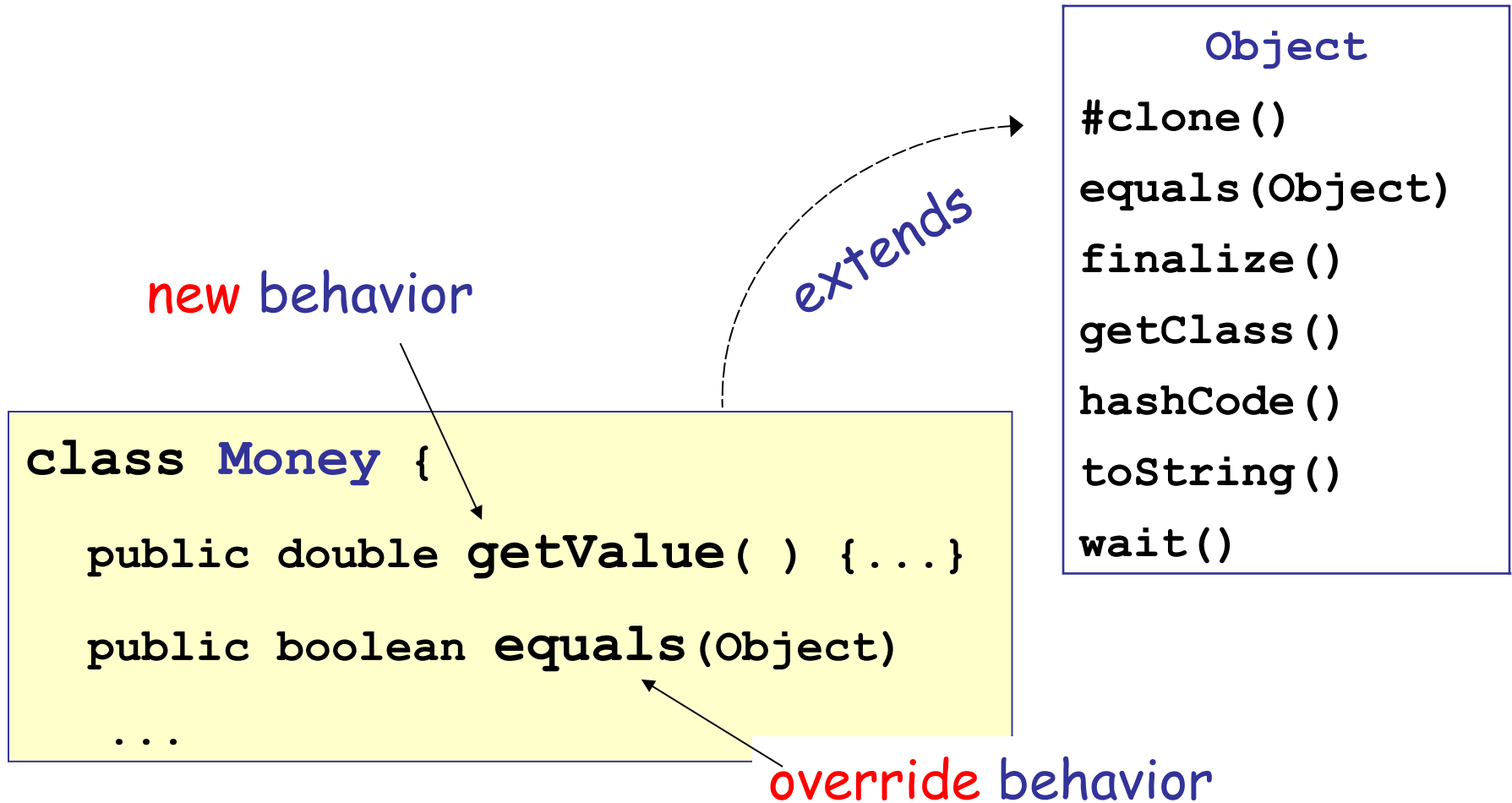
```
public void display(Number num) {  
    System.out.println("The value is "+num.intValue() );  
}  
  
display( new Integer( 10 ) );  
display( new BigDecimal( 3.14159 ) );
```

The value is 10

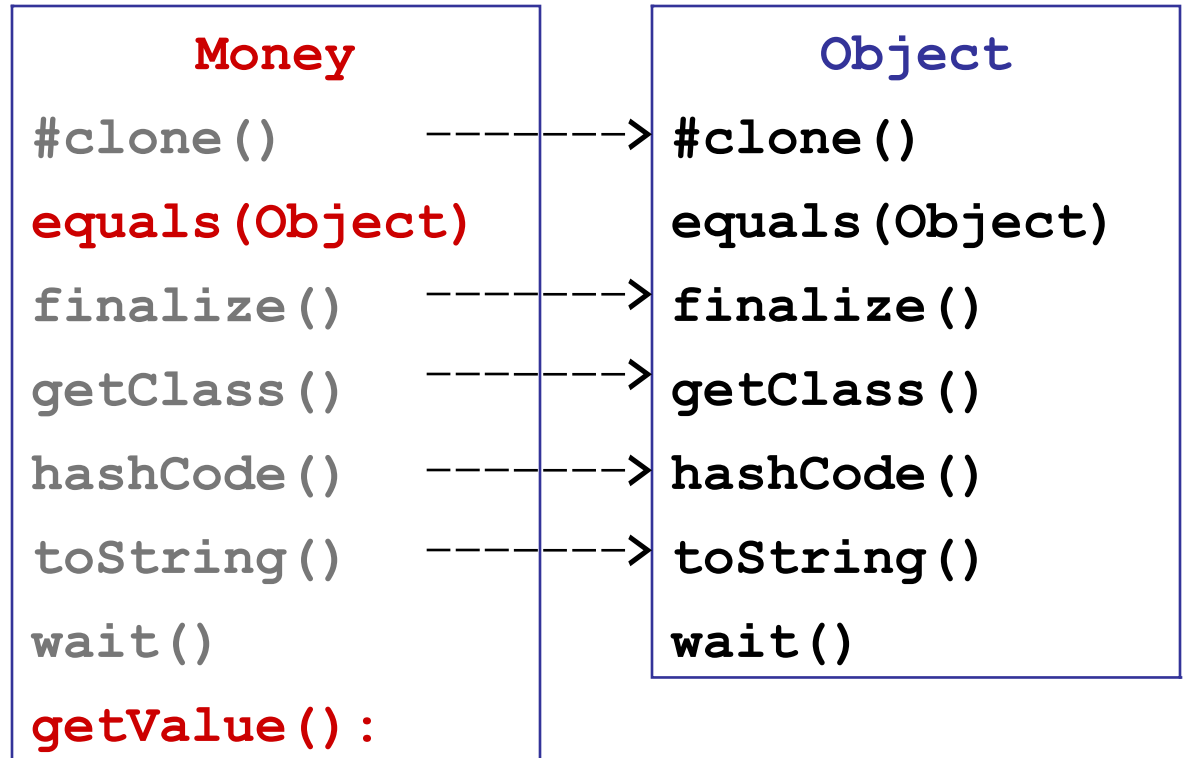
The value is 3

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 a superclass method:

```
public class Money {
    public int compareTo( Money other )
}
public class Coin extends Money {
    @Override // this tag is not required
    public int compareTo( Money other )
}
```

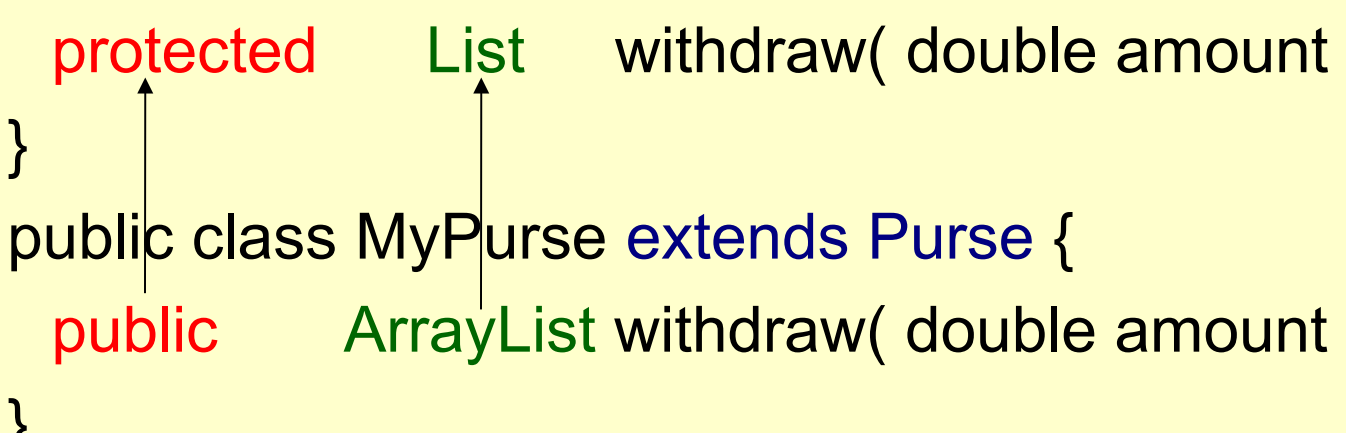
What Can Override Methods Change

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

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

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

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



New Method, not Override

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

```
public class Money {  
    public int compareTo( Money 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 Error: invalid "override" (wrong param)  
    public boolean equals( Money other ) {  
        return this.value == other.value;  
    }  
    // Typing error: new method "tostring" should be toString  
    @Override  
    public String toString() {  
        return "Money, money";  
    }  
}
```

If you write @Override, the compiler will warn you of misspelled "toString"

Two uses of @Override

1. In Java 5, @Override always 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 can define a **new method** with same name.

✓ final methods

Redefining final methods is not allowed.

Compile-time error.

Preventing Inheritance: *final class*

A "**final**" class cannot have any subclasses.

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

All "enum" types 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 Account {  
    // don't let subclasses change deposit method  
    public final void deposit(Money amount) {  
        ...  
    }  
}
```

final method

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

```
public class Coin extends Money {  
    // Error - override not allowed  
    public double getValue( ) { ... }  
}
```

Question: Does `Object` have any final methods?

Why should they be final?

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...

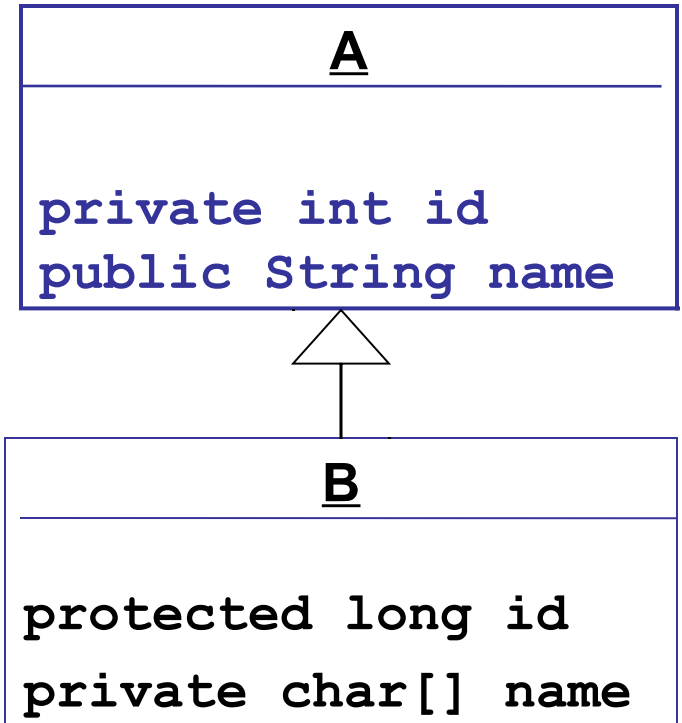
b1: B

long id = 1234567890

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

(hidden) int id = 0

(hidden) String name = "unknown"





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 is a subclass of **Number** .

Therefore, **Double** must also have a `longValue()`

Java Example

```
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?  
}
```