



# Introduction to Inheritance

---

James Brucker

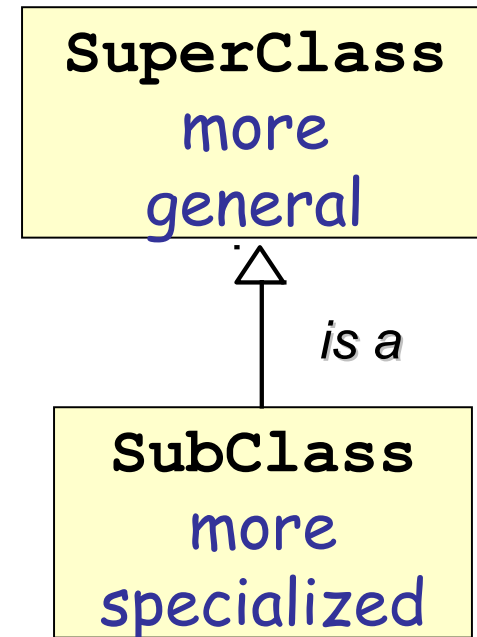
These slides cover only the basics of inheritance.

# 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

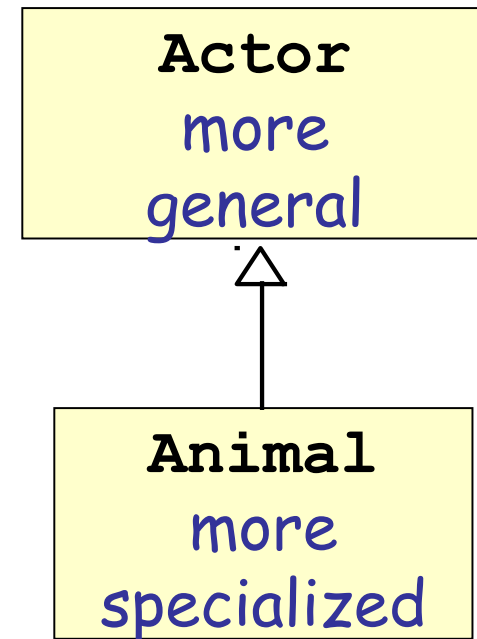
# Terminology

---

Different names are used for inheritance relationships.

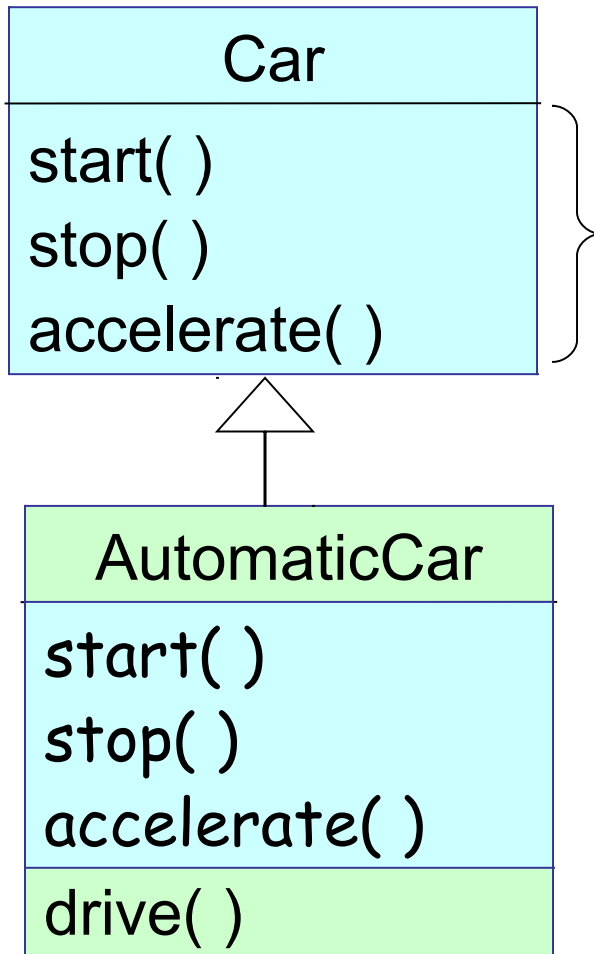
They mean *the same thing*.

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



# "Specializing" or "Extending" a Type

---



Consider a basic **Car**.

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

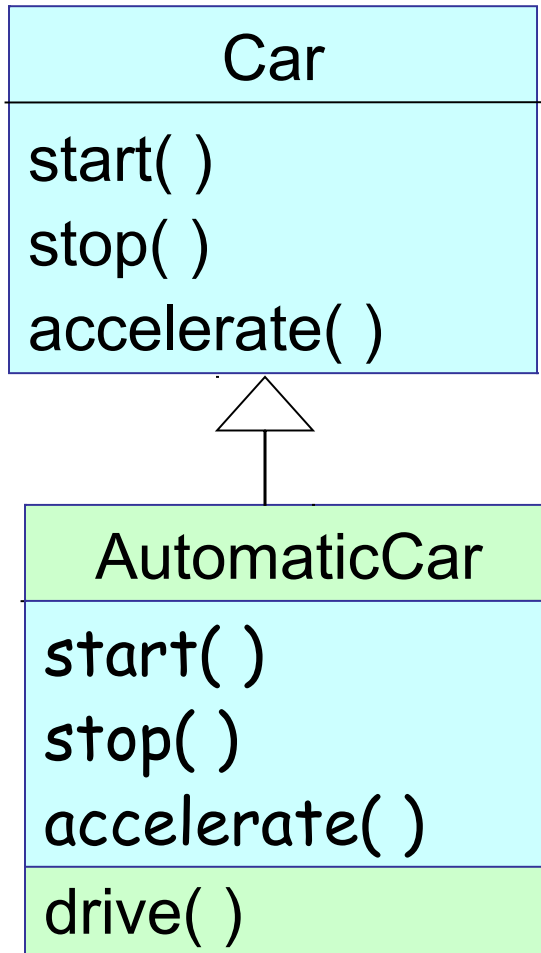
An **AutomaticCar** is a *special kind* of **Car** with automatic transmission.

**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**".

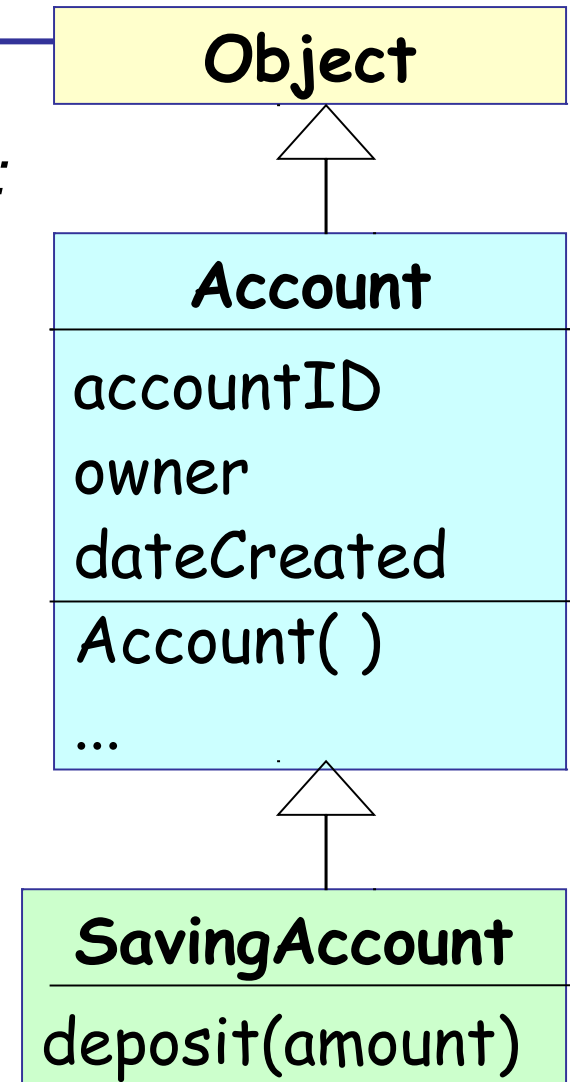
# What do you *inherit*?

A subclass *inherits* from its parent classes:

- ✓ attributes
- ✓ methods - even private ones.
- ✓ cannot access "**private**" members of parent, but they are inherited

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

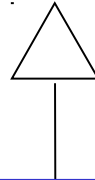
Any method that **Object** has, every class has.



# 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.

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



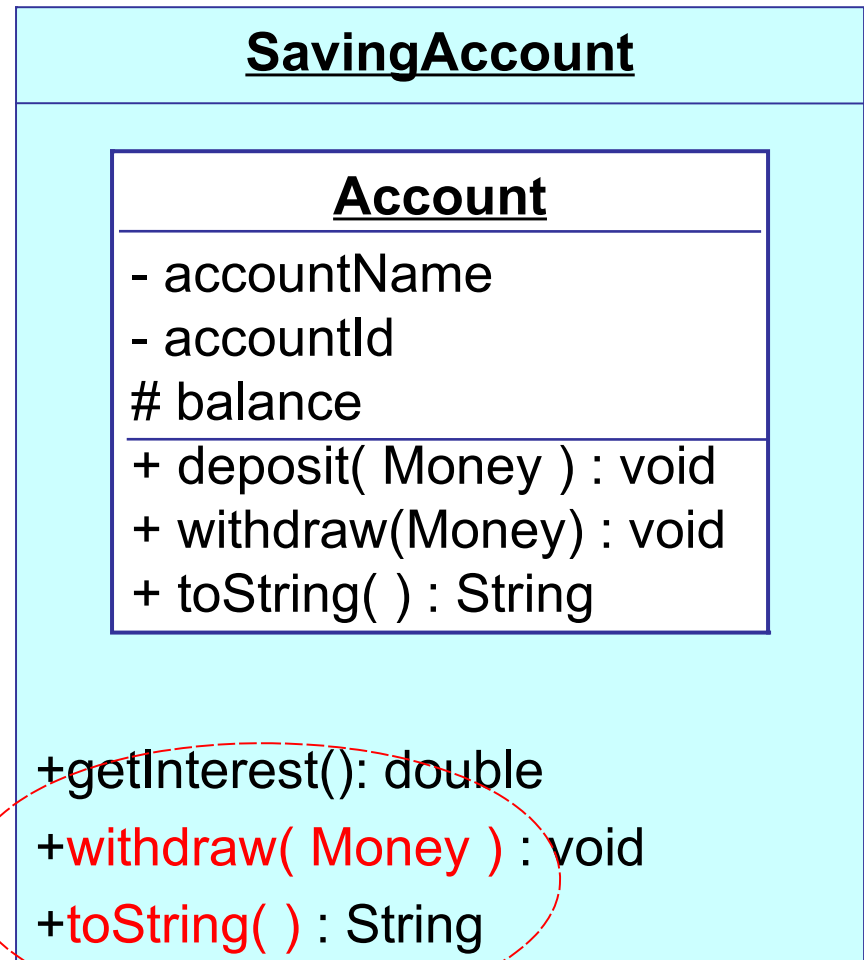
# Interpretation of Inheritance (2)

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) 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** accountId value 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:

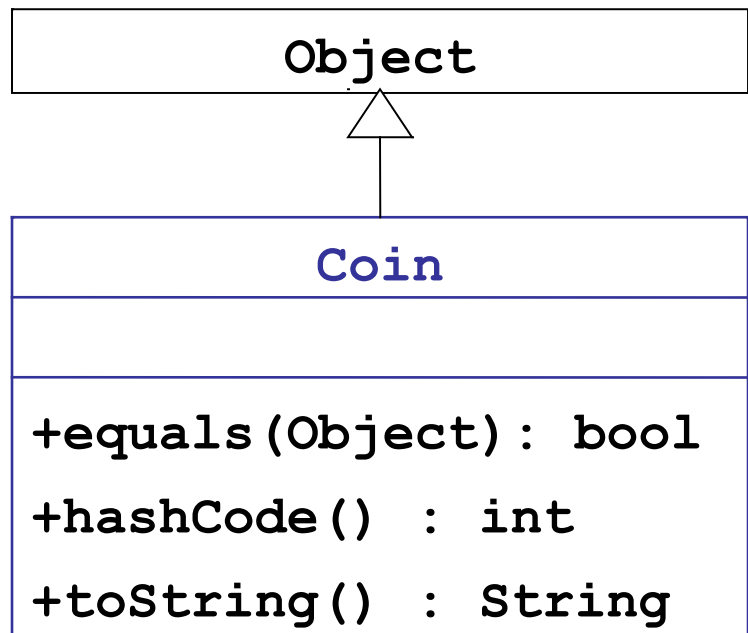
(1) inherit them

(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.

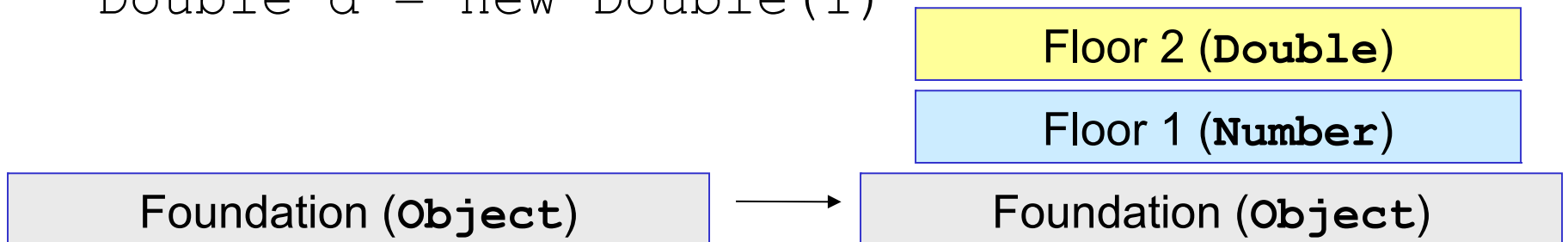
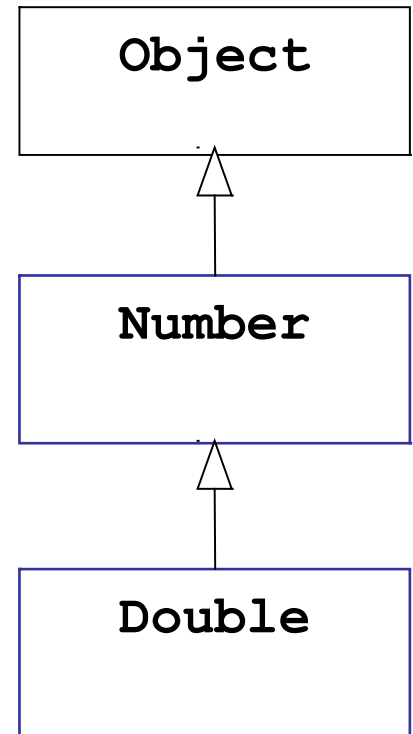
# Constructors and Inheritance

To **build** a building...

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

Example: Double is subclass of Number

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



# Which Constructor Executes First?

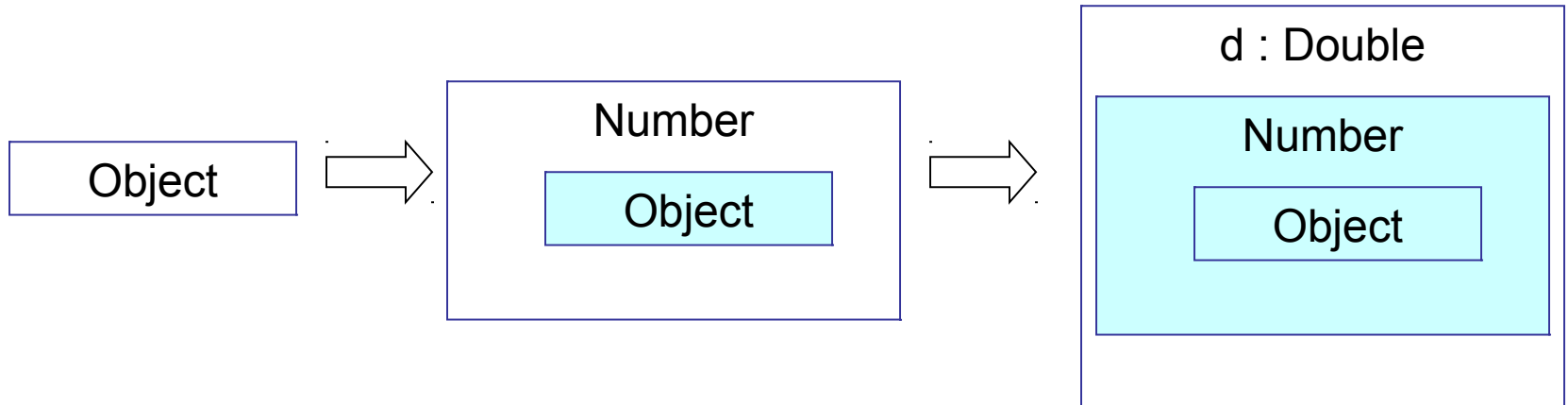
---

To *build* an object of the `Double` class...

- **first** you have to build the **foundation class** (`Object`)
- **then** build the **1st subclass** (`Number`)
- **then** build the **2nd subclass** (`Double`)

Example:

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



# Calling a Superclass Constructor

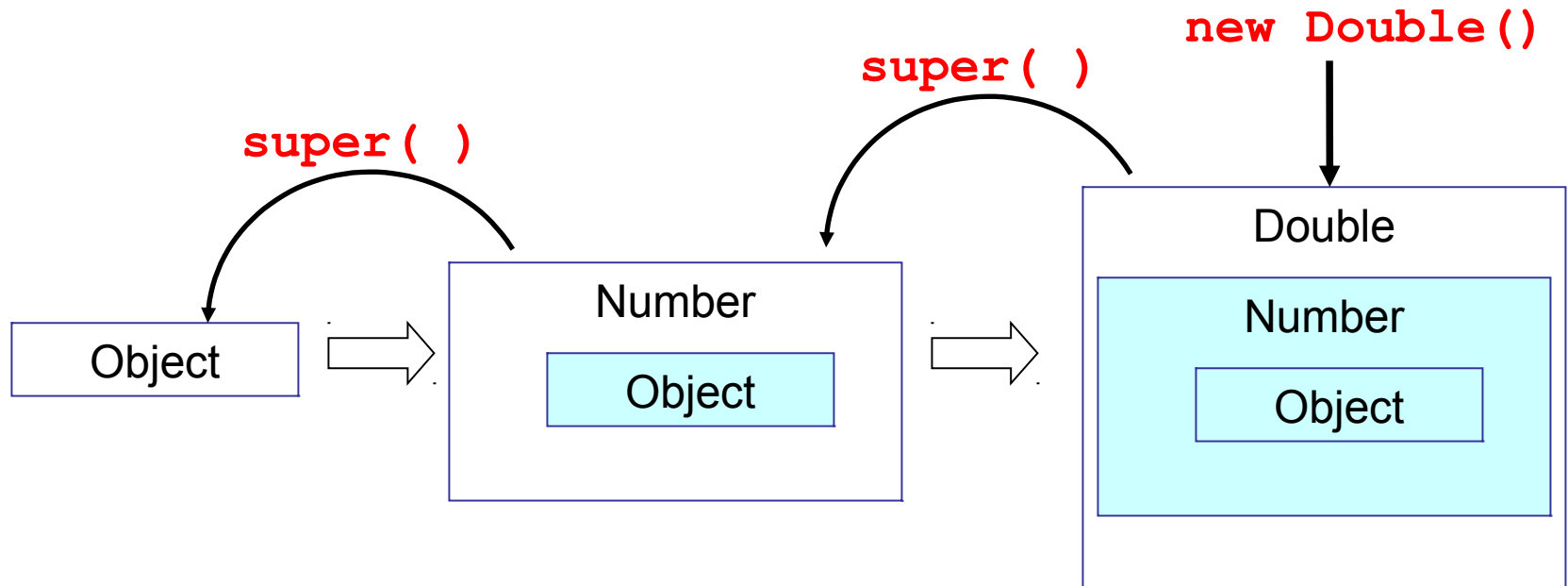
---

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

Example:

```
Double d = new Double(2.5);
```

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



# Try It!

---

Write 3 classes with a "default" constructor.  
Each constructor prints "Creating a new xxx"

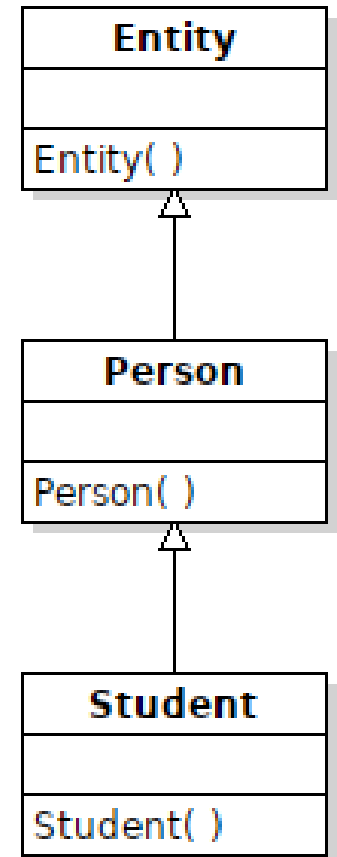
`Entity`

`Person` - subclass of `Entity`

`Student` - subclass of `Person`

`Main` - create a `Student` & print it

**What is printed?**





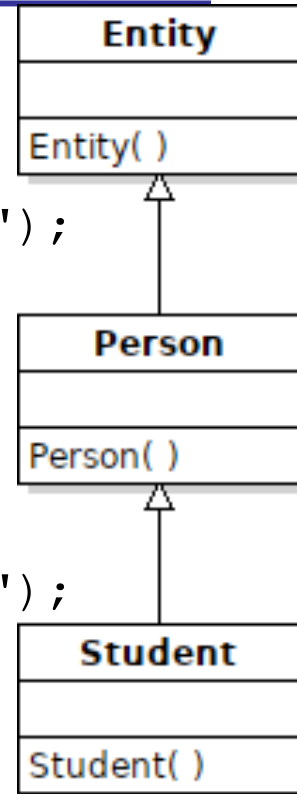
# Starter Code

```
public class Entity {
    /** Constructor for a new Entity */
    public Entity( ) {
        System.out.println("Creating a new Entity");
    }
}

public class Person extends Entity {
    /** Constructor for a new Person */
    public Person( ) {
        System.out.println("Creating a new Person");
    }
}

//TODO Write Student class

public class Main {
    public static void main(String[] args) {
        Student s = new Student();
        System.out.println("Student is "+s );
    }
}
```



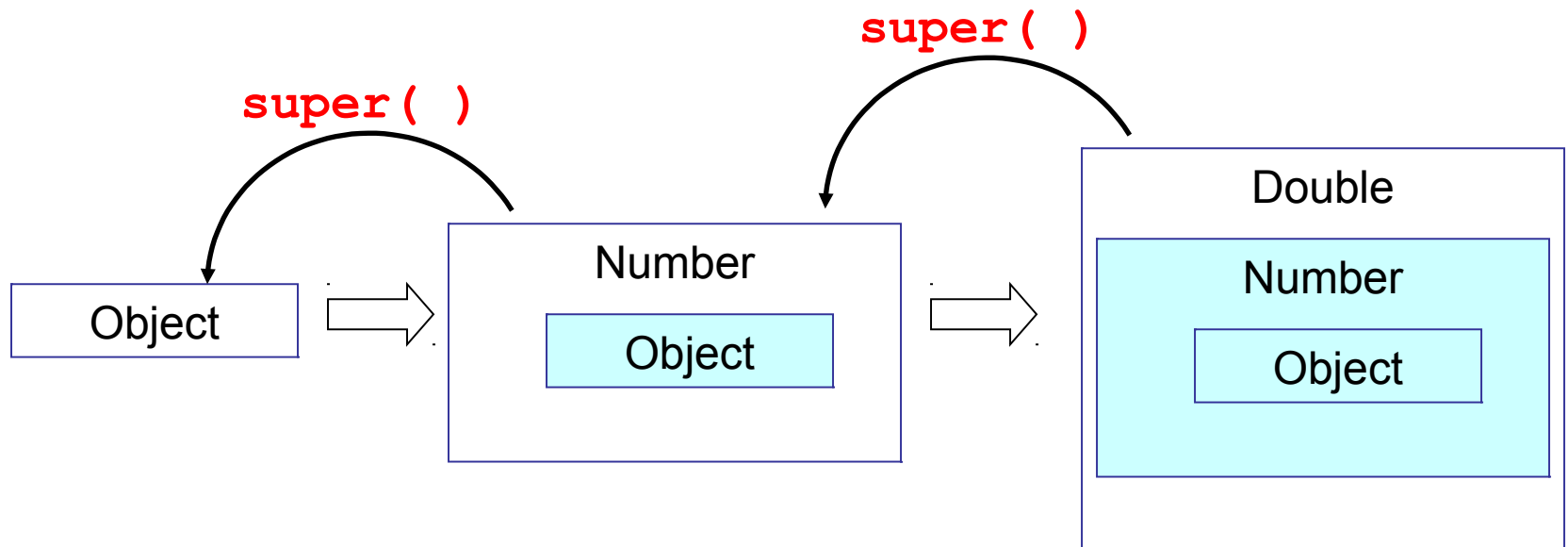
# Calling a Superclass Constructor

---

Each subclass **must invoke its subclass constructor** to "build" the superclass object.

**2 ways** to do this:

- **implicitly** - Java compiler inserts call to `super()`.
- **explicitly** write `super( )` in constructor code to invoke super-class constructor



# Implicit call to superclass Constructor

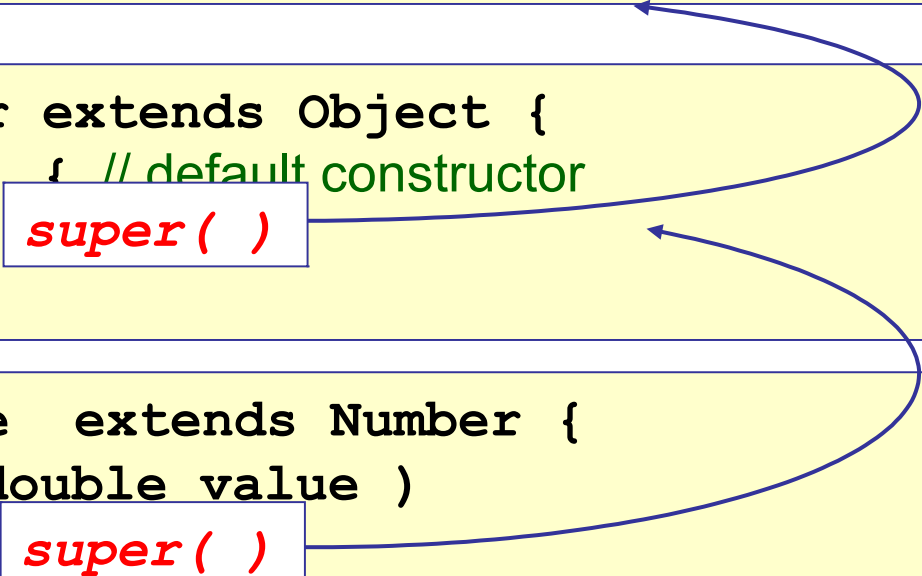
---

- 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 (no params)

```
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 ;  
    }  
}
```

The diagram consists of three yellow rectangular boxes containing Java code. Blue arrows originate from the `super ( )` call in the `Number` class and point to the `/* constructor for Object class */` comment in the `Object` class. Another blue arrow originates from the `super ( )` call in the `Double` class and points to the `// default constructor` comment in the `Number` class. The `super ( )` calls in the `Number` and `Double` classes are highlighted with white boxes with red text.


# 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 Person extends Entity {
    protected String name;
    public Person( String name ) {
        this.name = name;
    }
}
```

```
public class Student extends Person {
    public Student( String name, long id) {
        super( name ); // means: Person(name)
        this.id = id;
    }
}
```



# Add Constructor Parameters

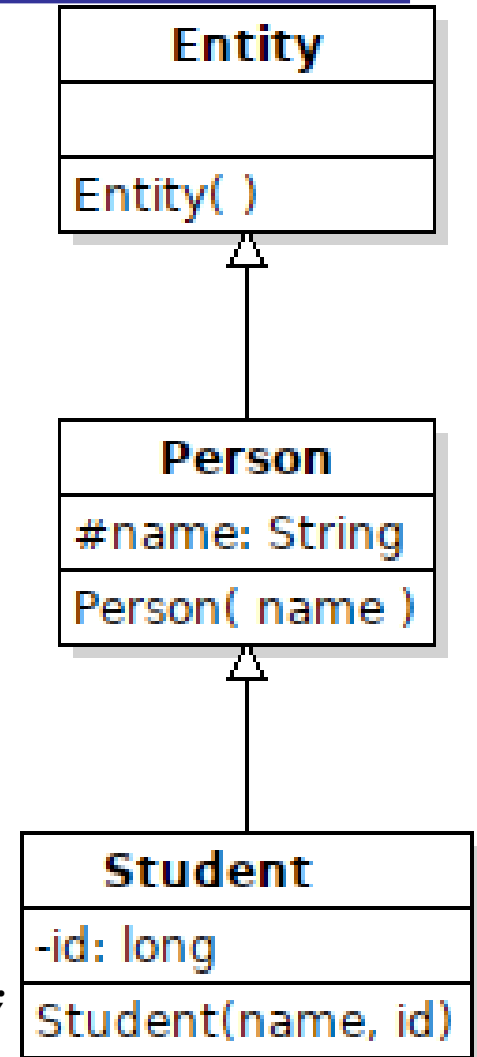
Person constructor requires a name.

Student constructor requires name & id.

```
public class Person extends Entity {
    protected String name;
    public Person(String name) {
        this.name = name;
    }
}

public class Student extends Person {
    // DO NOT REDEFINE "name" here
    private long id;
    public Student(String name, long id) {
        this.name = name; // access from Person
        this.id = id;
    }
}

// In Main.main:
Student s = new Student("Joe Hacker", 60111222L);
```



**What happens?**

## Error in automatic call to super( )

---

The Java compiler issues an **error** message:

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

The compiler added an implicit call to `super()`, but `Person` does not have a default constructor.

```
public class Student extends Person {  
    public Student(String name, long id) {  
        implicit call to super( )  
        // initialize Student attributes  
        this.id = id;  
    }  
}
```

What's the solution?

# Assign Responsibility!

---

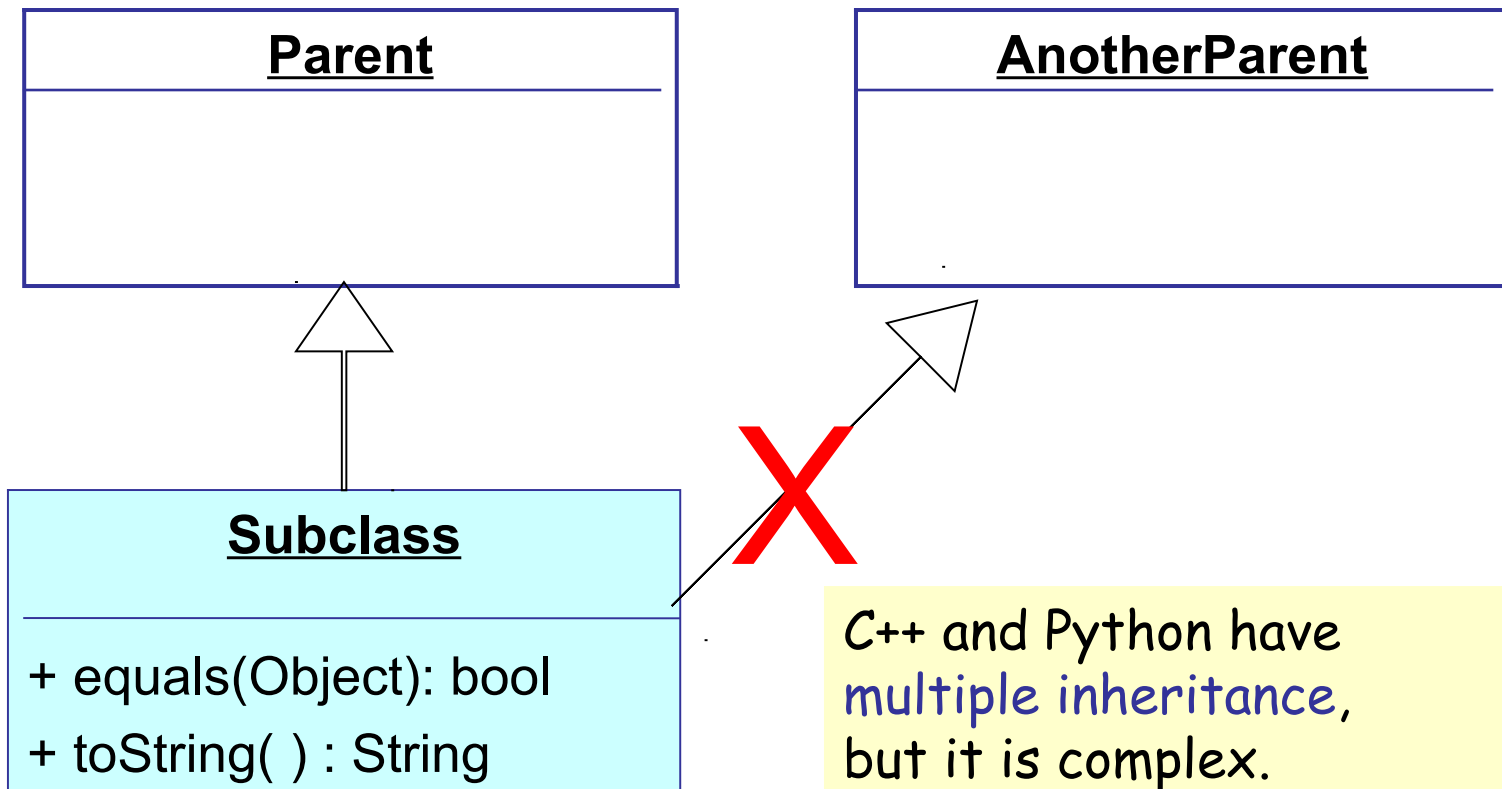
```
public class Student extends Person {
    public Student( String name, long id) {
        super( name ); // explicit call to super
        this.id = id;
    }
}
```

- The **name** attribute belongs to Person.
- The Person class should be **responsible** for setting the name, getting the name, testing the name, etc.
- Its good **encapsulation**. (name can be `private`, too.)

# A Class has only **One** Parent Class

---

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

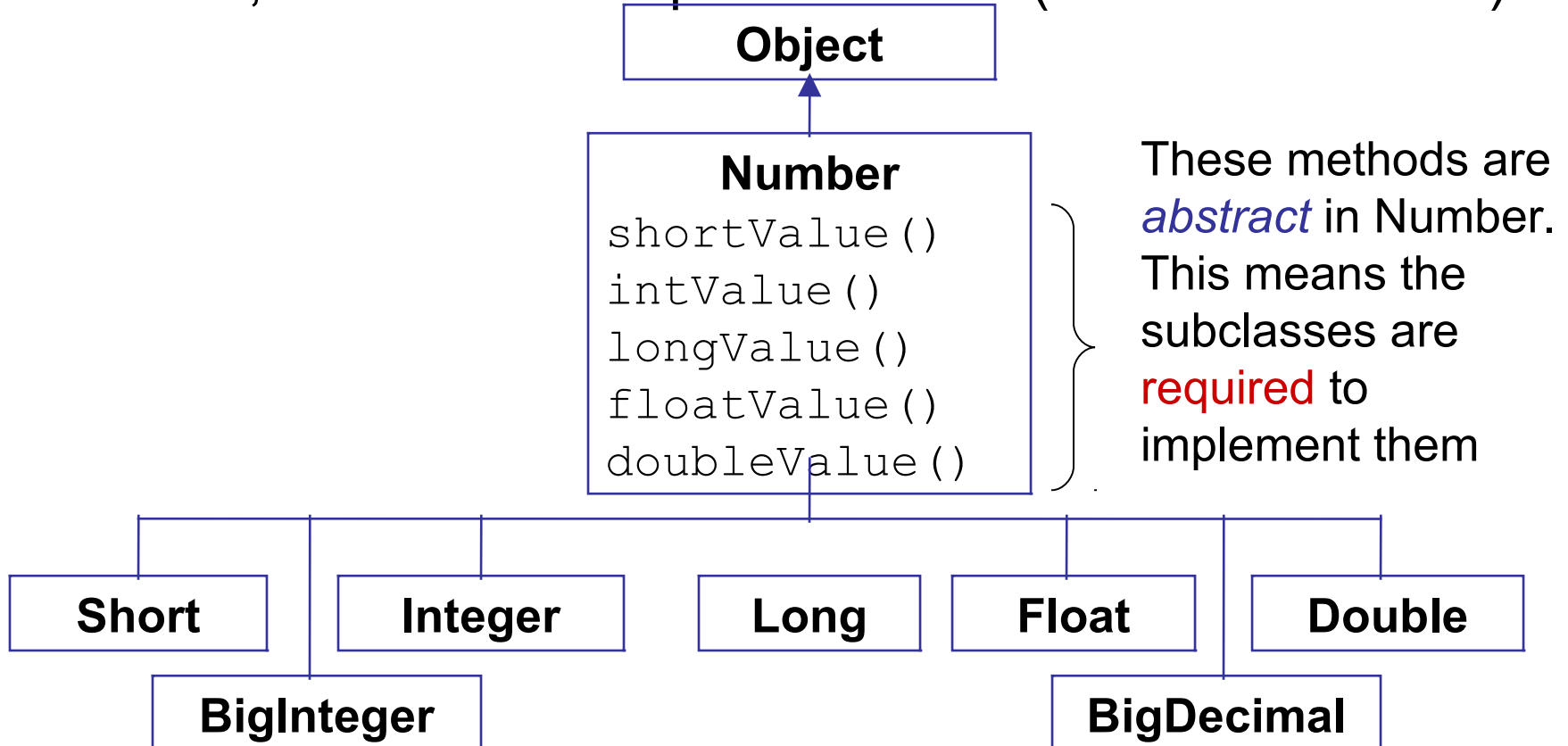




# Number: parent of numeric classes

---

- 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 ) );
```

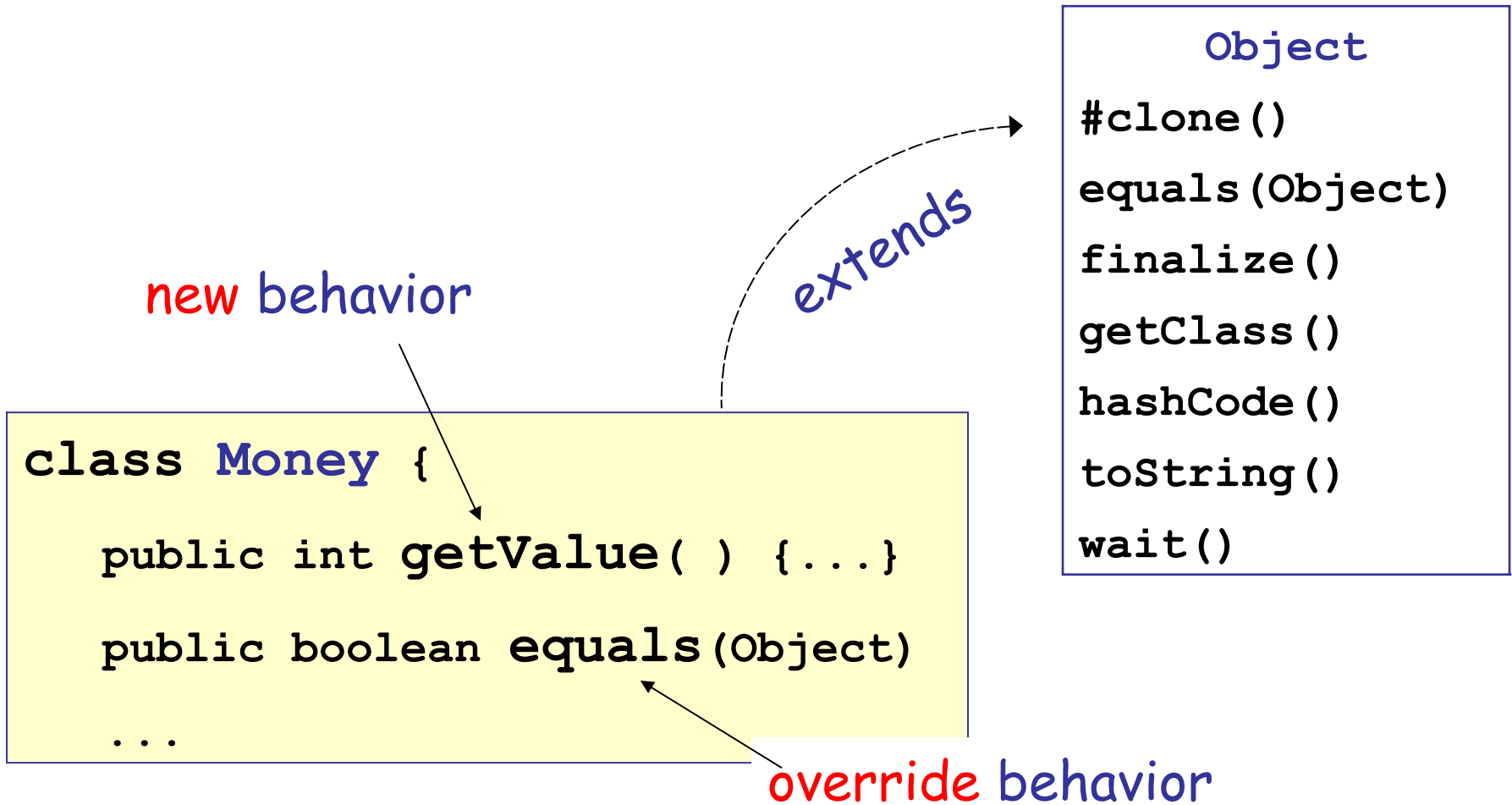
The value is 10

The value is 3

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

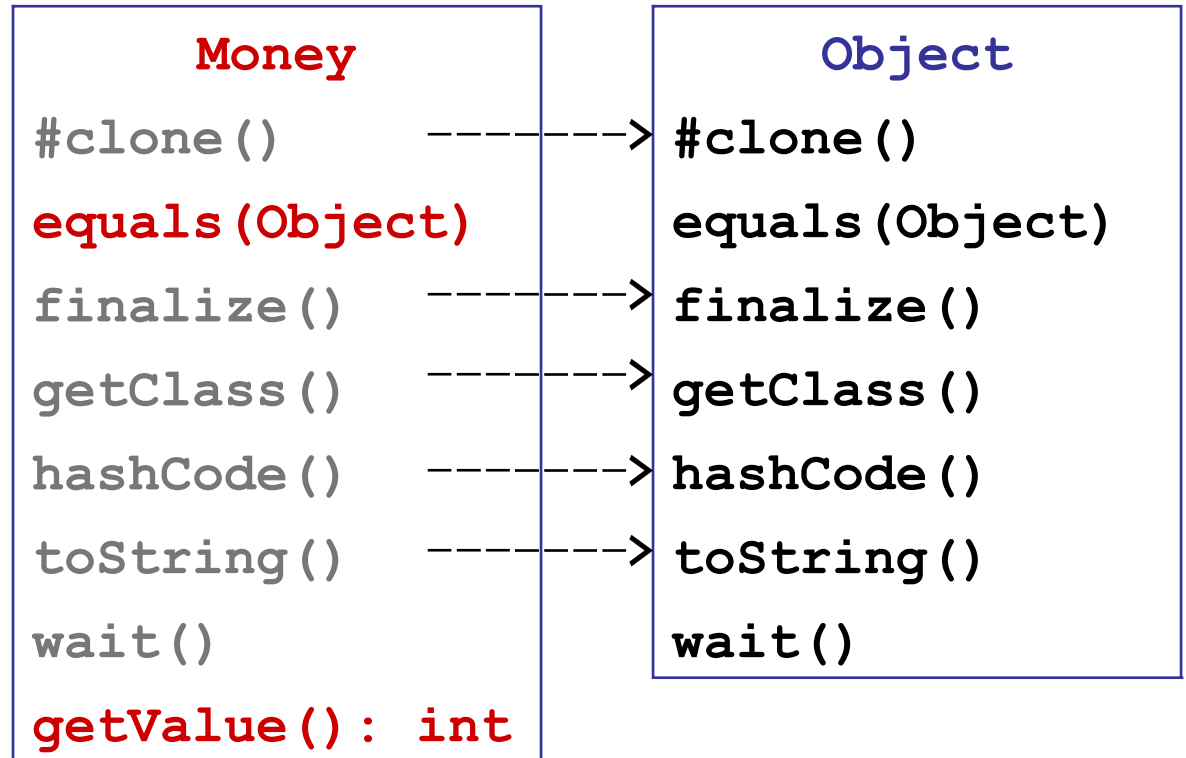
# Inherited Methods

---



# Inherited Methods

---



# Summary: Override vs New Method

---

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

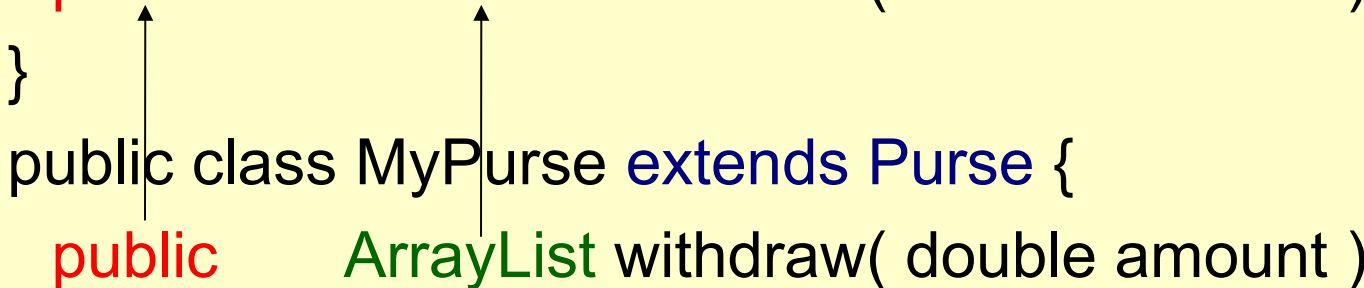
---

Override method can change **2 things** in 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 change to a method's parameters 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** errors.

```
public class Money {  
    @Override // Compile-time ERROR: invalid "override"  
    public boolean equals( Money other ) {  
        return this.value == other.value;  
    }  
    // Typing error: accidentally define a new method "toString"  
    // but no error from compiler because it does not have @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 forbidden. Compile-time error.

# Preventing Inheritance: *final class*

---

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

All methods in a **final** class are **final**.

All "enum" types are final.

**Final classes:** String, Double, Float, Integer, ... are final.

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

# Try It!

---

Try to define a subclass of String. What happens?

```
public class MyString extends String {  
    public MyString(String text) {  
        super(text);  
    }  
    // all methods are inherited  
}
```

# 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 in Object?

---

Question:

Does **Object** have any final methods?

The Java API (Javadoc) will show which methods are 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...

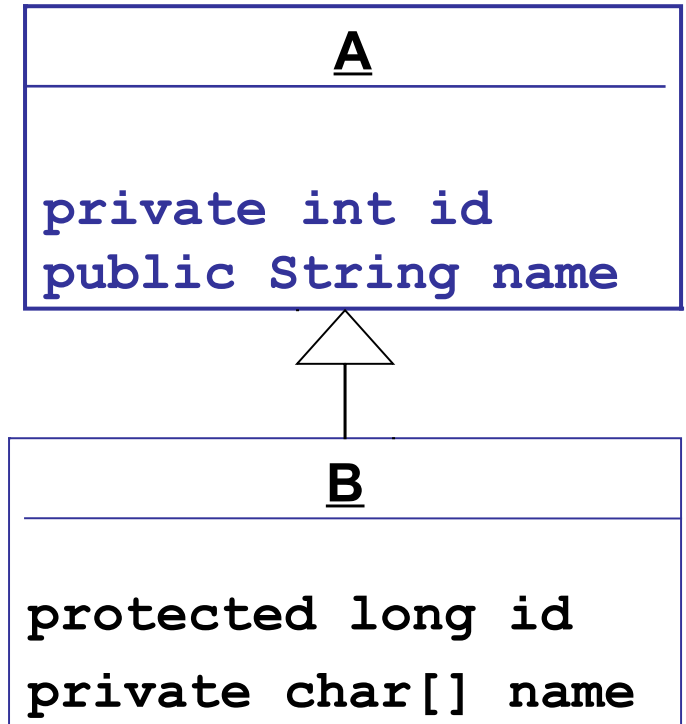
b1: B

long id = 1234567890

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

(hidden) int id = 0

(hidden) String name = "unknown"







# 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



# Review Questions

---

# 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*

# What visibility can you write here?

```
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?
```

# Visibility in override methods

---

**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.

# Can you change the return type?

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

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

**The Test:** does polymorphism work?



# Can you change the parameter type?

---

**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 {
    public boolean withdraw( long? amount ) { ... }
```

# Overriding Methods: parameters

---

**Answer:** If you change the parameter type(s), then you defining a new method, not overriding a superclass 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()
```

# Can a subclass invoke methods from the superclass?

---

**Q:** Can a subclass access a 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;
    }
}
```

# Redefining Attributes

---

A subclass can declare an **attribute** with the **same name** as an attribute 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;  
}
```

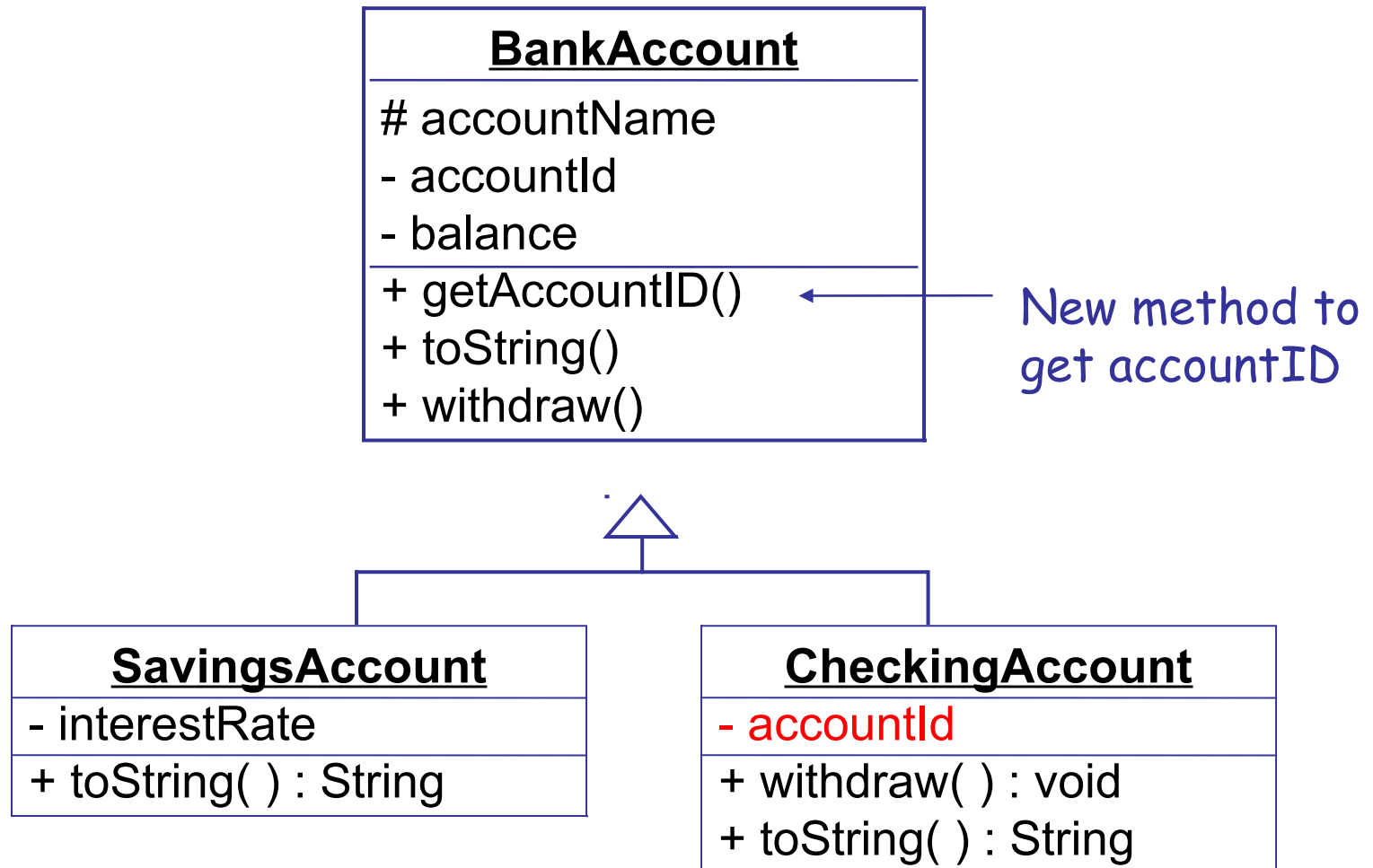
```
public class SavingAccount  
    extends BankAccount {  
    private String accountId;  
}
```

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

# Redefining Attributes

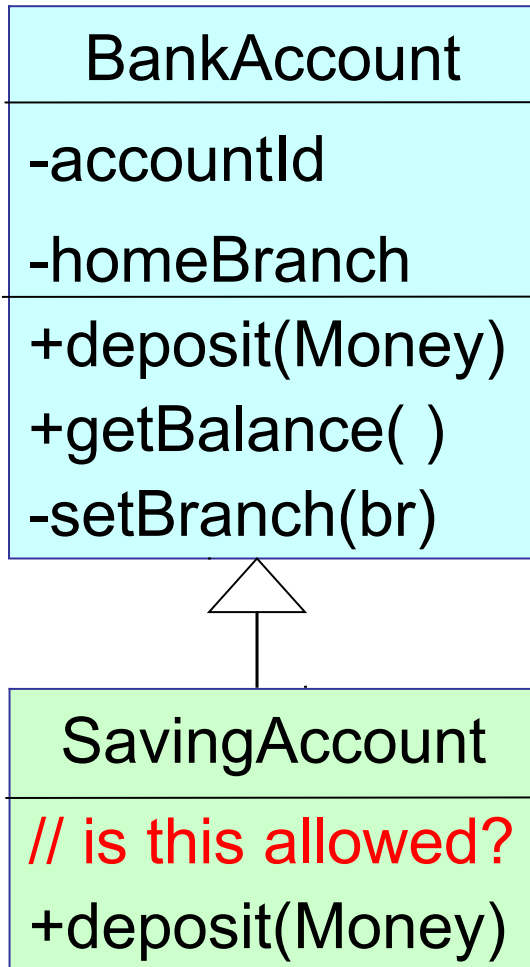
---

The new BankAccount hierarchy is:



# What does a subclass inherit?

---



1. Does SavingAccount have an accountId (private in BankAccount)?
2. Does SavingAccount have a setBranch( ) method?
3. Can SavingAccount define its own deposit method?
4. Can SavingAccount define **its own** homeBranch?
5. Is there any way for BankAccount **to prevent** SavingAccount from overriding the deposit method?

# 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** (for 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** is a subclass of **Number** .

Therefore, **Double** must also have a `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?  
}
```