Day 4

COMP1006/1406

Summer 2016

M. Jason Hinek
Carleton University

today's agenda

- assignments
  - questions about assignment 2
- a quick look back
  - constructors
  - signatures and overloading
  - encapsulation / information hiding
  - inheritance
- polymorphism
- abstract classes

last time...

constructors

a constructor is called when an object is created

- has no return type (not even void)
- must have the same name as the class
- are not method

we can call another constructor using `this()` provided it is the first line of the constructor (constructor chaining)

`this` is also a reference to the current object and is used when paramater names are the same as attribute names

last time...

signatures and overloading

A method or constructor **signature** consists of

- the **name** of the method or constructor
- the **input parameters** of the method or constructor (number, type and order)

Java identifies methods and constructors by their signatures.
This allows for method and constructor **overloading**.

- overloading allows multiple methods/constructors to have the same name as long as their signatures are different (return types do not matter!)

last time...

encapsulation / information hiding

good oop design makes use of encapsulation (information hiding)

hide the internal representation of the state and provide an interface to use the data

- make all attributes (data) `private` or `protected`
- provide public **getters** to view data
- provide public **setters** to modify data
- provide other public methods that operate on the data

last time...

inheritance

the `extends` keyword allows a class to extend or inherit from another class

- if you don't extend a class you automatically extend `Object`
- a class can explicitly extend only one class
- when you extend a class you receive things (attributes/methods) from the parent class

now let's look at

more stuff!

Java's Object class

```
public class Object{
    /* no attributes */

    /* single constructor */
    public Object(){}

    /* 11 methods */
    public String toString(){...}
    public int hashCode(){...}
    public boolean equals(Object obj){...}
    ...
}
```

- `java.lang.Object`
- this is Java's root class
  (its basic non-primitive type)

inheritance

Let's run the following simple class;

```
public class Student {
    /* attributes */
    public String name;
    public int id;

    public static void main(String[] args){
        Student s = new Student();
        System.out.println(s);
        System.out.println(s.toString());
        System.out.println(s.hashCode);
        System.out.println(s.equals(s));
    }
}
```

inheritance

Let's run the following simple class;

```
public class Student {
    /* attributes */
    public String name;
    public int id;

    public static void main(String[] args){
        Student s = new Student();
        System.out.println(s);
        System.out.println(s.toString());
        System.out.println(s.hashCode);
        System.out.println(s.equals(s));
    }
}
```

Why does this work?

inheritance

```
public class Student{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

when you compile this class it is automatically modified to inherit from the `Object` class

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

- Java keyword `extends` used for inheritance
- when we inherit from a class
  - we get all public/protected attributes from the parent class
  - we get all public/protected methods from the parent class
  - we get none of the constructors

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

- Java keyword `extends` used for inheritance
- when we inherit from a class
  - we get all public/protected attributes from the parent class
  - we get all public/protected methods from the parent class
  - we get none of the constructors
- we say that Student is an Object
  - this is the "is-a" relationship
- we say that Student has a String
  - this is the "has-a" relationship
  - this is class composition (not inheritance)

inheritance

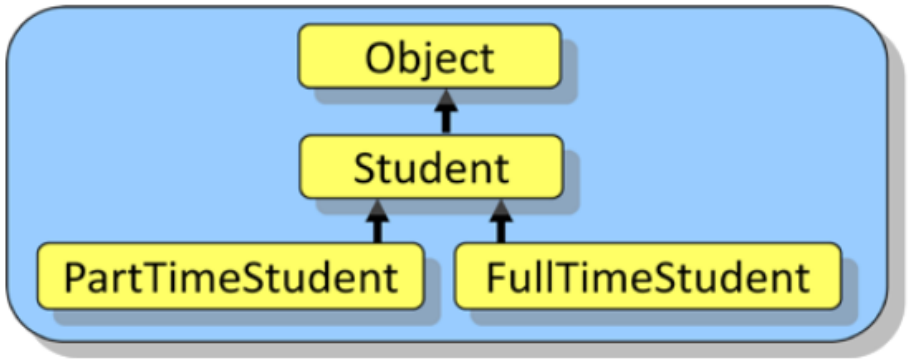
```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

- a class can only have one parent class
- every class, except `Object`, has exactly one parent class
- we get a hierarchy of classes
  - a family tree of classes

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

- a class can only have one parent class
- every class, except `Object`, has exactly one parent class
- we get a hierarchy of classes
  - a family tree of classes



inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

we say that

- Student is a child class of Object
- Student is a subclass of Object (direct subclass)
- Student is a derived class of Object
- Student is a descendent of Object

we say that

- Object is a parent class of Student
- Object is a super class of Student (direct super class)
- Object is a ancestor of Student

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

a Student object gets everything (state and behaviour) that an Object has in addition to whatever else we define in the Student class

this helps us reduce code!

inheritance

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;
    ...
}
```

what do we get from `Object`?

- `toString()`
- `hashCode()`
- `equals(Object o)`
- eight other methods that we will most likely not use

Are these useful?

Object's equals method

```
public boolean equals(Object obj)
```

- checks if both `this` and `obj` are the same
- returns `this == obj`
- this is almost certainly not what we want! (why?)

Aside: Java's equality operator `==` should only be used for primitive data types (and null) in most cases. `a == b` returns true if the value of the left hand side is the same as the value of the right hand side. But for objects, these values are references and not the data itself.

Object's toString method

```
public String toString()
```

- returns a string representation of the object
- the returned string looks something like
  - `Ball@5af660f` (for a Ball object)
  - `Ljava.lang.String;@462467e5` (for am array of Strings)
- this is almost certainly not what we want!

Object's hashCode method

```
public int hashCode()
```

- returns a hash code value for the object
- can think of this like an integer fingerprint of the object
  - different objects can have the same has code value
- we may want to use this

inheritance and method overriding

The methods that we inherit from Object may or may not be useful.

We can change the bahaviour of inherited methods though.
We do this through **method overriding**

```
@Override
public String toString(){
  return this.name + ", " + this.id;
}
```

this will redefine the `toString` method in the `Student` class.

- all student objects will run this code instead of Object's version
- this doesn't change Object's code, just the code in a Student

Note: the `@Override` is not part of the Java language. It is an annotation that the compiler to use to make sure that the method you are overriding is actually an inherited method.

inheritance and method overriding

```
public class Student extends Object{
    /* attributes */
    public String name;
    public int id;

    @Override                                   ← @Override is an annotation
    public String toString(){
        return this.name + ", " + this.id;
    }
}
```

method `overriding` allows us to redefine a parent's (or grandparent's) method definition. which method is executed?

- Java first looks in current class
- if method is not defined, look at parent class
- if method is not defined, look at parent class
- ...
- get method from `Object`

inheritance and method overriding

rules for **method overriding**

- signature must be identical (input parameter names can be different)
- return type must be the same or more restrictive
  - same if primitive
  - otherwise, same or a subtype (child class)
- modifiers must be the same or less restrictive
  - protected can be overridden as friendly or public
  - friendly cab be overridden as public
  - we cannot override private methods

inheritance and method overriding

inheritance lets us reduce write less code by inheriting methods from a parent or grandparent class

what happens when we need to make a change to the inherited method (overriding) but most of the code is still the same? we don't want to rewrite the entire method body of the parent.

the keyword `super` is a reference to the parent. we can use it to access parent attributes and parent methods.

```
public class Student extends Object{

    @Override                                    ← @Override is an annotation
    public String toString(){
        if( this.name == null ){
            return super.toString();
        }else{
            return this.name + ", " + this.id;
        }
    }
}
```

inheritance and constructors

by default, the first thing that any constructor we write does is call the zero argument constructor of its parent class

if want a different parent constructor called we need to explicitly call it using `super`

Take not that

- `super(...)` must be called on the first line of the constructor
- `this(...)` must be called on the first line of the constructor
- we cannot call both `super()` and `this()` in a constructor
- we can only access a direct parent's constructor.
  There is no `superduper`!

inheritance and constructors

let's see some examples...

inheritance

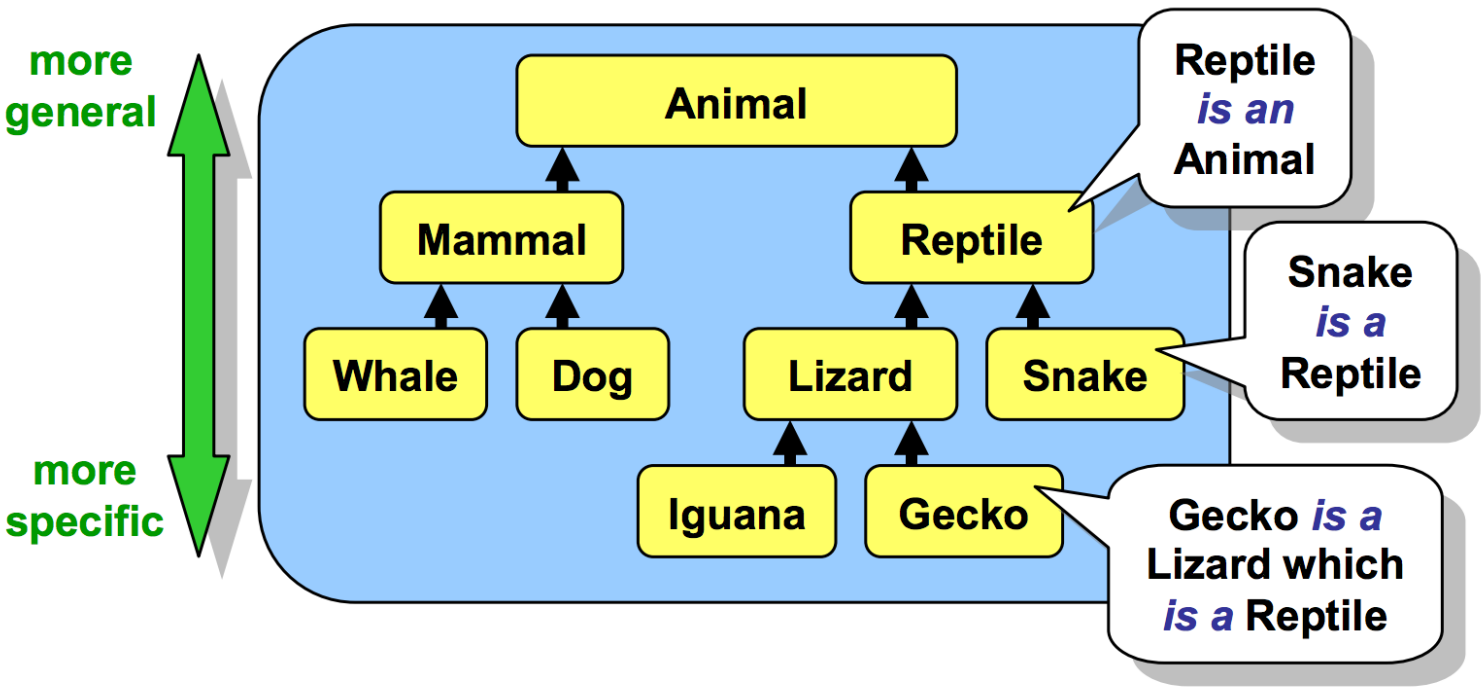
when the "things" that we are modelling in our programs have a natural hierarchical structure, inheritance it lends itself nicely to OOP and inheritance

- we can have a root class that is very general (animimals)
- root class has some children that are both animals but also very different from each other
- we continue creating new classes that become more specific (specific types of dogs)

inheritance

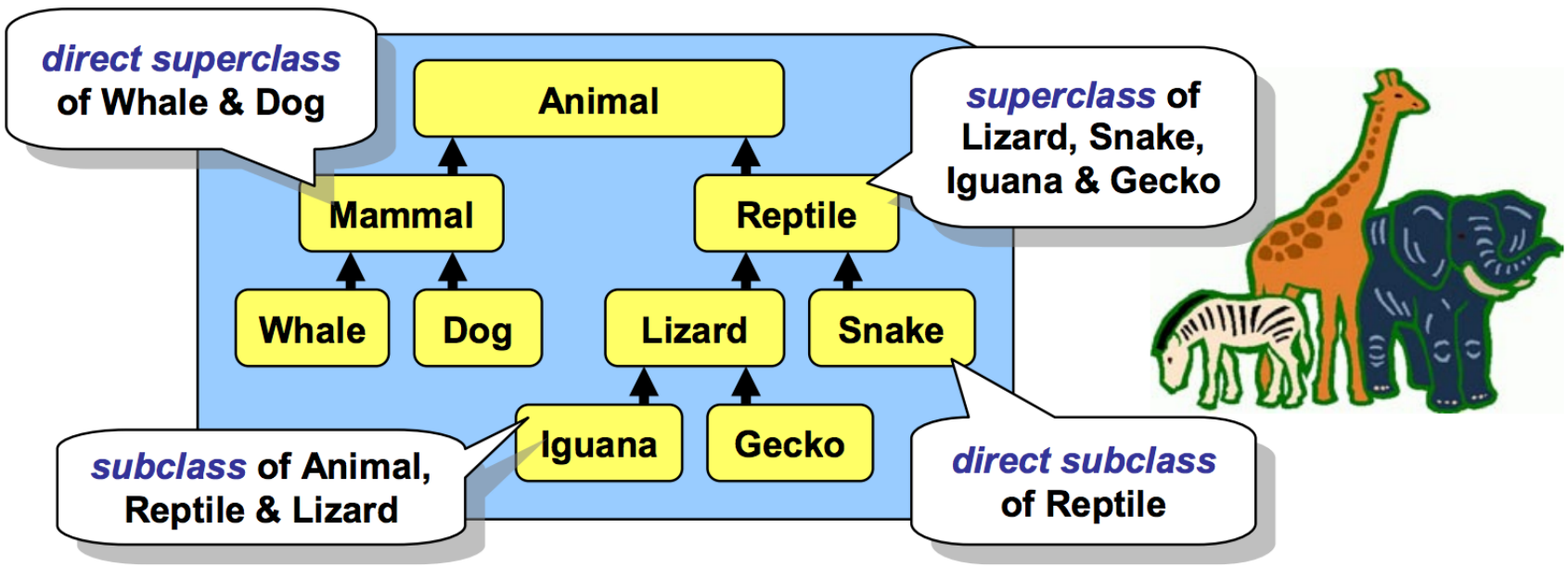
- from Dr. Lanthier's notes

  `http://people.scs.carleton.ca/~lanthier/teaching/COMP1406/Notes/COMP1406_Ch4_ClassHierarchiesAndInheritance.pdf`



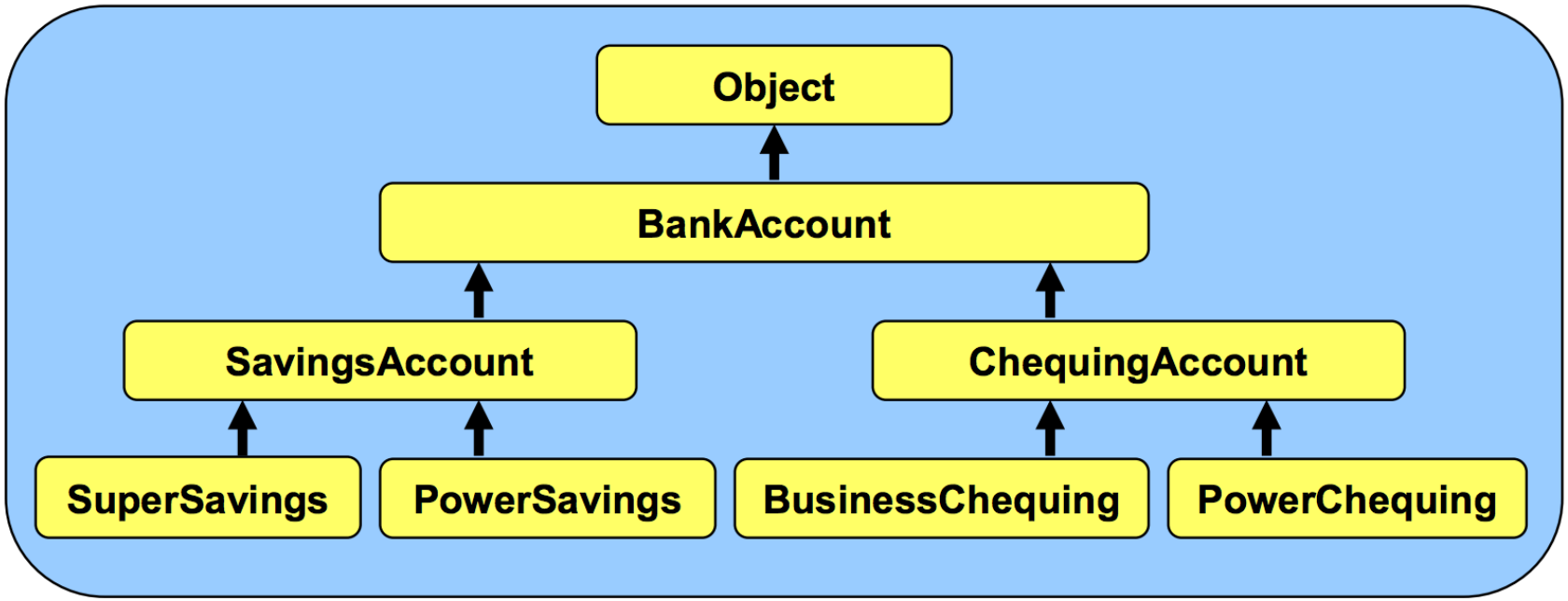
inheritance

- from Dr. Lanthier's notes



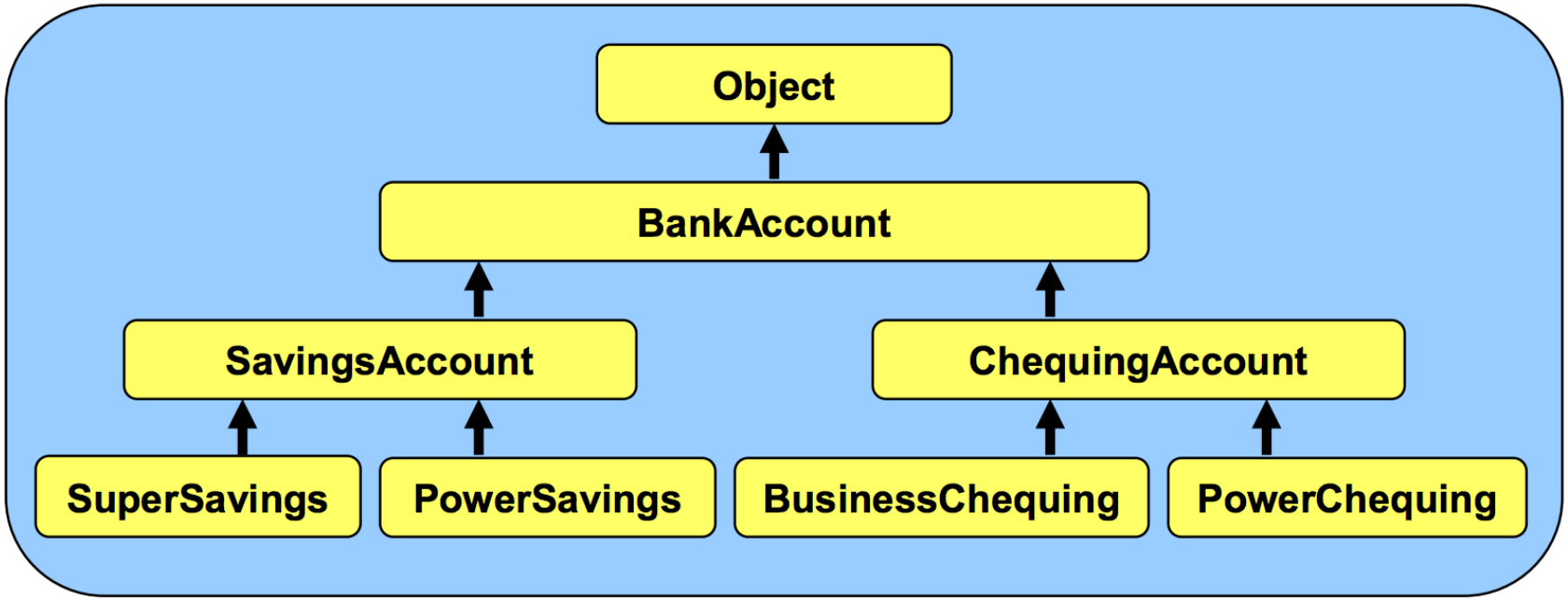
inheritance

- banking example



inheritance

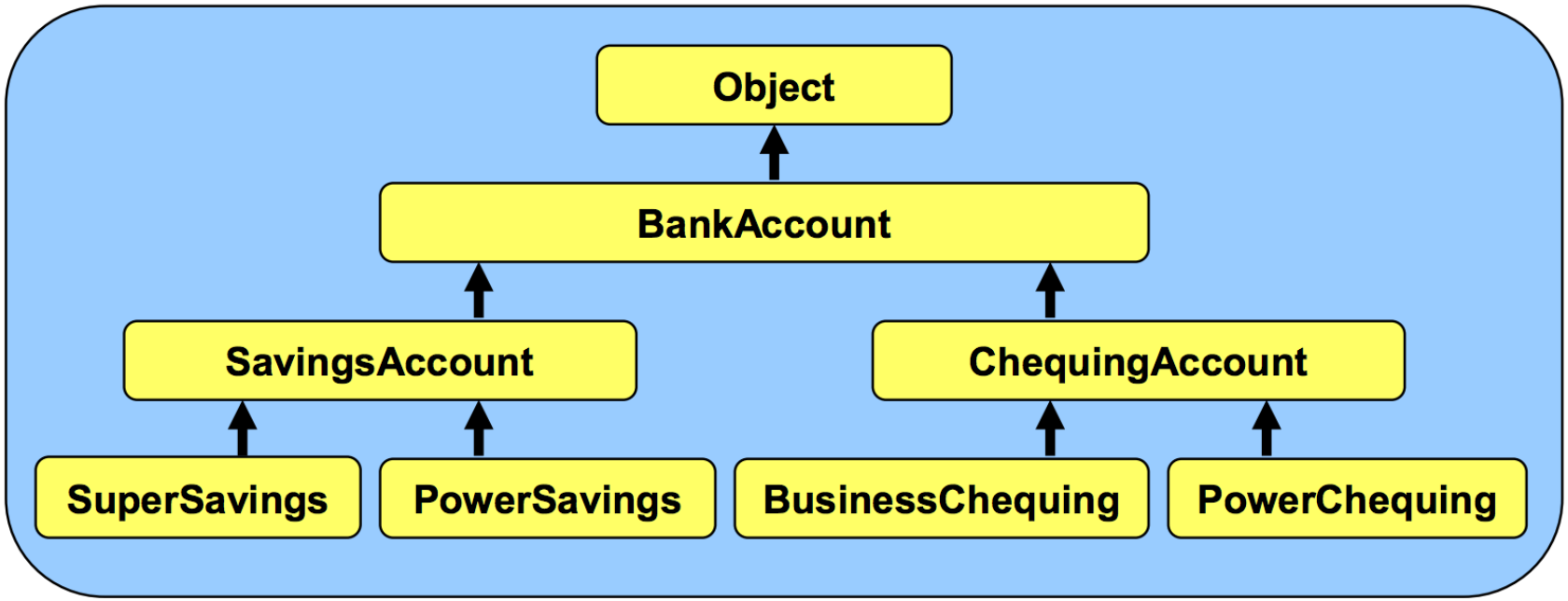
- banking example



so what? We have reused our code but how does this actually help?

polymorphism

- banking example



so what? We have reused our code but how does this actually help?

- polymorphism to the rescue!

polymorphism

- banking example

```
BankAccount b1 = new PowerSavings();
BankAccount b2 = new BusinessChequing();
BandAccount b3 = new ChequingAccount();
```

polymorphism allows us to declare a variable for some type (class) and populate it with an object of a different type (class) as long as it is a subclass of the variable type

a PowerSavings object `is-a` BankAccount object, so we are allowed to have a BankAccount variable that references it

`b1` is a BankAccount object, but the behaviour of it is defined by the PowerSavings class

polymorphism

polymorphism allows subclasses to define their own unique behaviour without losing the functionality of the superclass(es).

A polymorphic object is capable of executing specific behaviour based on its type.

Method overloading and overriding are examples of polymorphism.

- **early binding** or **static binding** is when method behaviour of an object is determined at compile time.
  - static, private and final methods are bound at compile time
  - the compiler knows exactly which method is being called
- **late binding** or **dynamic binding** or **dynamic dispatch** is when method behaviour of an object is determined at runtime
  - the compiler does not know which method is being called so it has to wait until runtime to determine this
  - `BankAccount b;` is this a BankAccount, a PowerSavings, a BusinessChequing, ...?

polymorphism

with **polymorphism**

- objects can act like other objects
  (very related objects though!)
- this simplifies code understanding
  - have one array of BankAccount objects that holds everything
- standardizes method naming
  - all BankAccount objects have similar behaviour defined by the BankAccount class
  - you can add a new kind of BankAccount by simply writing a new class that extends BankAccount and it will still have the same behaviour

now let's look at

abstract methods and classes

abstract methods

an **abstract method** allows us to declare a method without an implementation

```
public abstract String getName(int n);
```

the intention is for subclasses to override (define) the method

- forces the same method on child classes
- cannot be a `final` method (as these cannot be overriden)
- forces class itself to also be `abstract`

abstract classes

an **abstract class** cannot be instantiated.
a **concrete class** can be instantiated.

```
public abstract class AbstractPlayer{
    ...
}
```

the intention is to provide framework for child classes

- class cannot be instantiated
  (but is a valid data reference type)
- cannot be a `final` class (as these cannot be extended)
- no restrictions on what can be in an abstract class
  - attributes, methods, constructors
  - does not need to have `abstract` methods
- intention is for a descendant class to be `concrete`

abstract classes

why use `abstract` classes?

abstract classes

why use `abstract` classes?

- why use inheritance?

abstract classes

why use `abstract` classes?

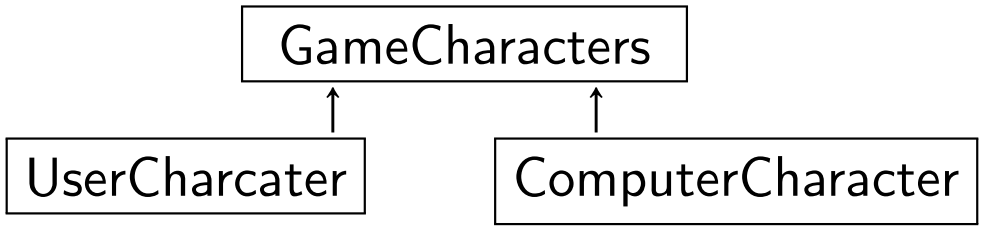
- why use inheritance?

GameCharacters

abstract classes

why use `abstract` classes?

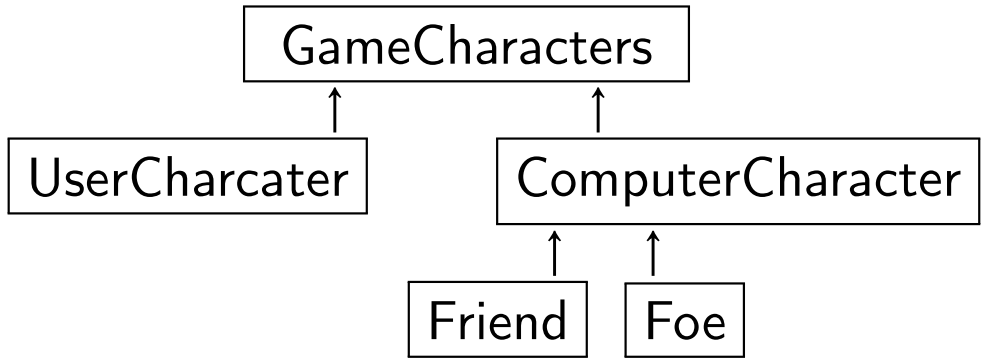
- why use inheritance?



abstract classes

why use `abstract` classes?

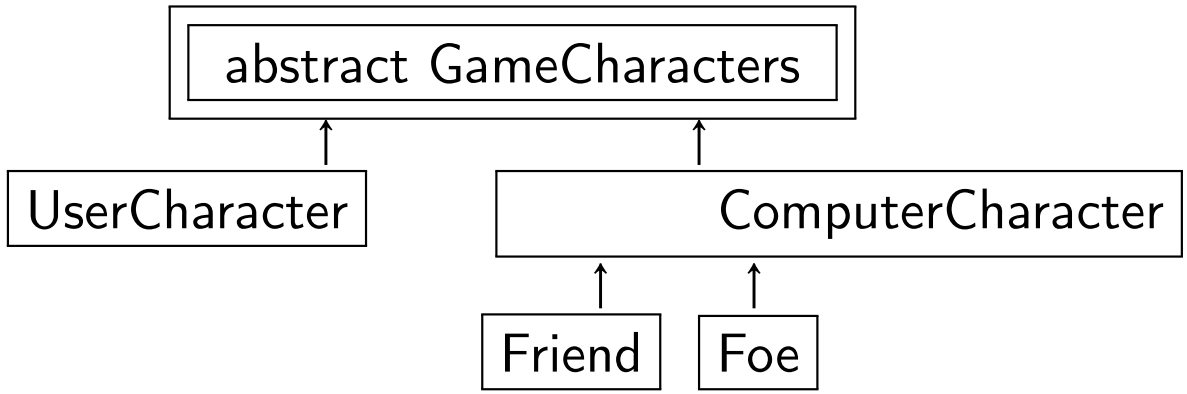
- why use inheritance?



abstract classes

why use `abstract` classes?

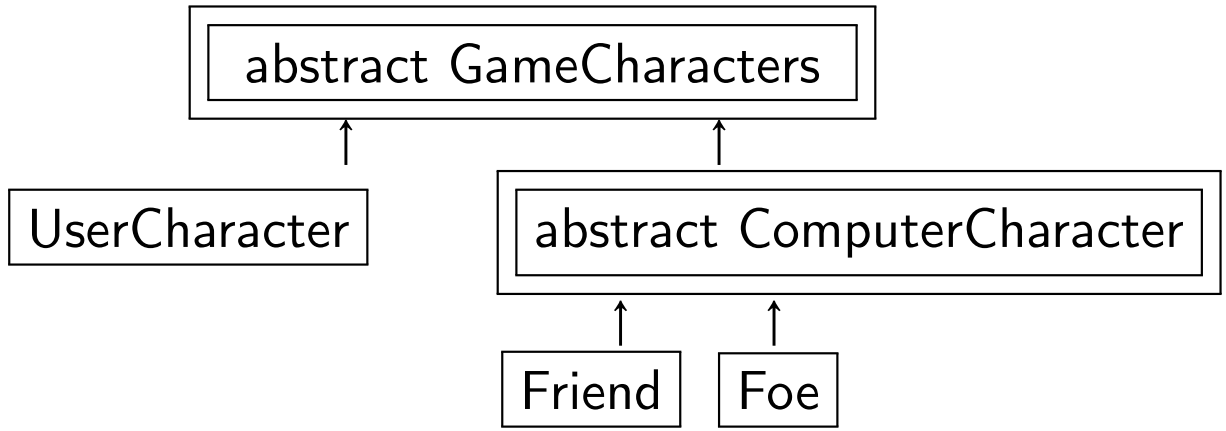
- why use inheritance?



abstract classes

why use `abstract` classes?

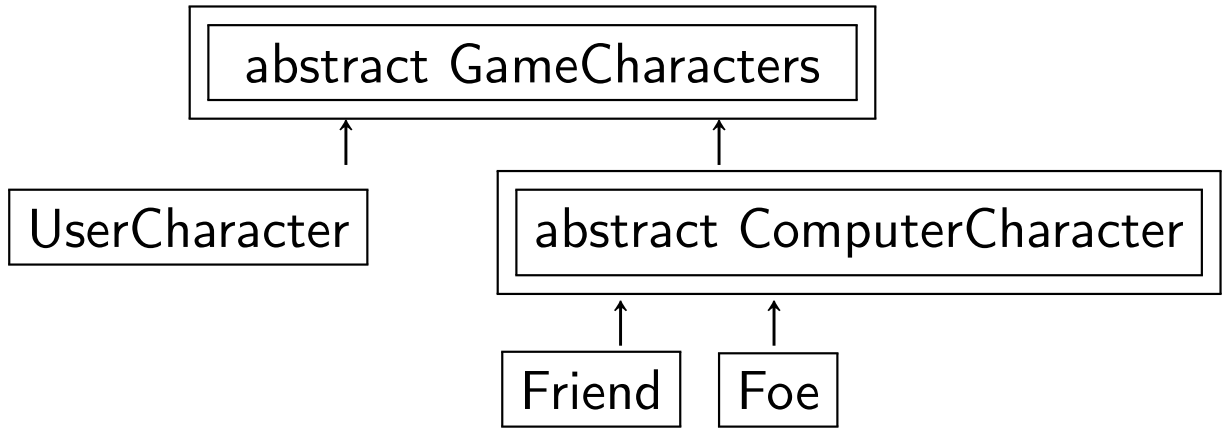
- why use inheritance?



abstract classes

why use `abstract` classes?

- why use inheritance?



- an abstract class can provide an interface or partial interface between implementation and users of code
- helps modularize code

overriding and hiding

inheritance can get tricky in Java...(and other languages)

- method **overriding**
  - `instance` methods are overridden
  - same method signature (name and argument list) and return type*
- method **hiding**
  - `static` methods are hidden
  - same method signature (name and argument list) and return type*
- attribute **hiding** (field **hiding**)
  - any attribute in the parent class with the same name is hidden
  - type of attribute does not matter
  - Avoid attribute hiding!
- method overriding/hiding leads to different behaviour

now let's look at

Interfaces

abstract classes and interfaces

Java abstract classes

- related classes `extend` the same `abstract` class
  - similar behaviour is implemented in the abstract class itself
  - different behaviour is implemented in overridden abstract methods
- you can only `extend` one single class
- all classes that `extend` a given abstract class are related
- abstract class can provide a contract between code and users of code

Java interfaces

- provide a contract between code and users of code
  - typically specifies how different code interacts with each other
- a class can `implements` any number of interfaces
- classes that `implements` a given `interface` can be very unrelated

abstract classes and interfaces

Java abstract classes

- often defines what you **are**
- usually a noun;

Java interfaces

- often defines what you **can do**
- usually an adjective; Comparable, Serielizable, Printable, etc
- can also define what you are (List, Set)

interfaces

```
public interface Printable{
  boolean sendToPrinter(String);
  boolean killPrintJob(int);
}
```

- can contain abstract method declarations (without definition)
  - are `public` by default (you can omit this)
  - implicitly `abstract` (you do not write this)
- can contain constant attributes
  - are `public static final` by default (can omit this)
- can have `default` methods and `static` methods (these have definitions) and `enum` types
- can `extend` any number of other `interfaces`
- is a valid data reference type
  - cannot be instantiated (like an abstract class)
  - we can have variables of this type

Comparable

consider the `Comparable` interface

```
public interface Comparable<Type>{
   int compareTo(Type other);
     // returns an integer X satisfying
     //   X < 0 if this is "less than" other
     //   X = 0 if this is "equal to" other
     //   X > 0 if this is "greater than" other
}
```

- declared like a `class`, using `interface` instead
- has a single method called `compareTo()`
  - by default all methods are `public abstract`
- uses generics (we'll revisit this more later)
  - allows us to treat types as parameters `<type>`
  - avoids the messiness we saw with `Object`'s `equals()`

Comparable

using the `Comparable` interface

```
public interface Comparable<Type>{
   int compareTo(Type other);
}
```

Comparable

using the `Comparable` interface

```
public interface Comparable<Type>{
   int compareTo(Type other);
}
```

- class that `implements` the `Comparable` interface either

Comparable

using the `Comparable` interface

```
public interface Comparable<Type>{
   int compareTo(Type other);
}
```

- class that `implements` the `Comparable` interface either
  - must override (define) the method `compareTo`, or

Comparable

using the `Comparable` interface

```
public interface Comparable<Type>{
   int compareTo(Type other);
}
```

- class that `implements` the `Comparable` interface either
  - must override (define) the method `compareTo`, or
  - must be `abstract`

Comparable

using the `Comparable` interface

```
public interface Comparable<Type>{
   int compareTo(Type other);
}
```

- class that `implements` the `Comparable` interface either
  - must override (define) the method `compareTo`, or
  - must be `abstract`
- `public class SomeClass implements Comparable<T>{ ...`

  `public interface SomeInterface extends Comparable<T>{...`

Comparable

```
public class Student implements Comparable<Student>{
  private String name;
  private int    id;

  @Override
  public static int compareTo(Student other){
     if(other == null){
        return 1;
     }
     return this.getID() - other.getID();
  }

  ...

}
```

```
Student s = new Student("cat", 12);
Student t = new Student("dog", 7);
System.out.println( s.compareTo(t) );
```

abstract...

- **abstract methods**
  - a method declared without a definition
  - `public abstract int foo(String[] in);`
  - forces the class to be abstract as well
  - cannot be `final`
- **abstract classes**
  - cannot be instantiated
  - may or may not contain abstract methods
  - are valid reference types and can be subclassed
  - cannot be `final`
- **concrete classes**
  - all methods (declared or inherited) must be defined
  - can be instantiated (all objects other than arrays are instantiations of concrete classes)
  - is a valid data reference type

final...

- **final attributes**
  - value cannot be changed once it is defined
  - must be defined in constructor or initialization block
  - primitive data types, strings and immutable data types are **constants**
- **final methods**
  - cannot be overridden
  - cannot be `abstract`
- **final classes**
  - cannot be extended
  - cannot be `abstract`

access modifiers...

modifier	class	package	subclass	world
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
none (default)	✓	✓	✗	✗
`private`	✓	✗	✗	✗

- everything is accessible from within the class
- a class in the same package has access to everything except private members
- a subclass has access to public and protected members
- everyone else only has access to public members