Inheritance

- inheritance defines an “is-a” relationship between classes

```
public class Apple extends Fruit {
  …
}
```

  - an apple is a (kind of) fruit

- subclasses inherit everything – instance variables and methods – *except* constructors
  - even `private` things, though they cannot be accessed directly
  - new access modifier: `protected` allows only the class and its subclasses to access

Inheritance

Subclasses –

- *can* add new elements (instance variables and methods)
  - a new method has a different header (name and/or number/type of parameters)
- *can* redefine (override) or extend methods
  - same header, new body
  - to extend, also invoke superclass version
- *must* define one or more constructors (in most cases)
  - constructor should first call superclass constructor, then initialize only the instance variables for its own class
- *cannot* redefine instance variables
- *cannot* remove instance variables or methods already defined

Subclasses

When defining a subclass, you can: (choose all that apply)

add instance variables that are not part of the superclass	10 respondents	**100 %** ✓
add methods that are not part of the superclass	9 respondents	**90 %** ✓
redefine superclass instance variables so they have a different type or purpose in the subclass	3 respondents	30 %
redefine superclass methods so they behave differently in the subclass	5 respondents	**50 %** ✓
remove instance variables that are part of the superclass		0 %
remove methods that are part of the superclass		0 %

you can change method bodies but not method headers or variable declarations

you can change runtime elements but not compile-time elements

declarations (variables, method headers) are like clothes – they define the outward appearance
  → the compiler goes by the clothes: if it looks like a duck, great! it can be asked to quack

method bodies are the real identity under the clothes – they define what happens if you ask the object to do something
  → the runtime system asks for a quack, but it’s the real thing underneath the clothes that determines what exactly that quack sounds like

Inheritance

`class B extends A` means –

- B has every instance variable A has
- B has every method header and body A has

(even if declared `private` in A – B has it, it just can't access it directly)

B *does not* inherit A's constructors.

access modifier `protected` allows only the class and its subclasses to access

In addition –

- B can add new instance variables
- B can add new method headers and bodies
- B can redefine a method of A – same header, different body

But B cannot take away anything A has or change types.

```
public class A {

  private int x_;

  public A ( int x ) {
    x_ = x;
  }


  public void set ( int x ) {
    x_ = x;
  }

  public int get () {
    return x_;
  }
}
```

```
public class B extends A {

  private int y_;

  public B ( int x, int y ) {
    super(x);
    y_ = y;
  }


  public int get () {
    return 2*super.get();
  }

  public int getOther () {
    return y_;
  }
}
```

B inherits x_ and adds y_
B inherits set
B redefines (overrides) get
B adds getOther

Inheritance

- an object is like an onion, with each class in the inheritance hierarchy describing a layer
- top-level class is at the core

- `this` refers to the current layer of the onion
- `super` refers to the next layer in

Syntax

For each of the following tasks, indicate whether this, super, or neither is required for carrying out the task. (Choose "neither" if this or super could be used but are not required.)

`this`, `super` are both valid but not required

`this` is valid but not required

accessing a protected instance variable declared in a superclass	[Choose]	neither
accessing an instance variable hidden by a local variable or parameter with the same name	[Choose]	this
extending the superclass version of a method instead of replacing it	[Choose]	super
calling another method in the same class	[Choose]	neither
calling a superclass constructor	[Choose]	super
calling another constructor in the same class	[Choose]	this

Inheritance

- `this` refers to the current layer of the onion
  - allows reuse of constructor bodies within the current layer
  - used to disambiguate between instance variables and parameters in the current layer with the same name
    - avoid by using _ for instance variables
  - it doesn't allow duplicate methods or instance variables
- `super` refers to the next layer in
  - necessary to construct the inner layers of the onion
  - used to call the version of the current method defined in the superclass layer
  - it doesn't move definitions around

Example

A bank account has an account number, an owner (a name), and a balance. All three values can be retrieved, and the owner can be changed. Money can be deposited into or withdrawn from the account, but the balance can't drop below 0.

A checking account is a kind of bank account. It has an account number, an owner (a name), and a balance. All three values can be retrieved, and the owner can be changed. Money can be deposited into or withdrawn from the account, but the balance can't drop below 0. You can also write checks on the account.

A savings account is a kind of bank account. It has an account number, an owner (a name), a balance, and an interest rate. All four values can be retrieved, and the owner can be changed. Money can be deposited into or withdrawn from the account, but the balance can't drop below 0. At the end of every month, the interest accumulated over that month is added to the balance.

- write classes `BankAccount`, `CheckingAccount`, and `SavingsAccount` with the elements and functionality specified, utilizing inheritance as appropriate

Inheritance

Inheritance is often talked about as a way to reuse existing classes or code – but while this often occurs, it is not why inheritance should be used.

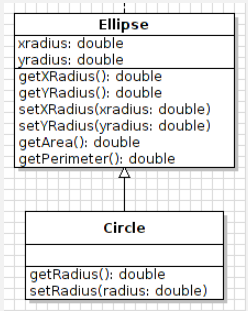
Create subclasses (only) when **both** –

- "is a", "is a kind of" language makes logical sense, **and**
- everything inherited from the superclass makes sense for the subclass

- this is known as the Liskov Substitution Principle
- introduced by Barbara Liskov in 1987
- she won the 2008 Turing Award for work leading to the development of object-oriented programming

(the Turing Award is kind of like the Nobel Prize for computer science – it's a big deal)

Inheritance and the Liskov Substitution Principle



Should Circle extend Ellipse?

Circle "is a kind of" Ellipse...

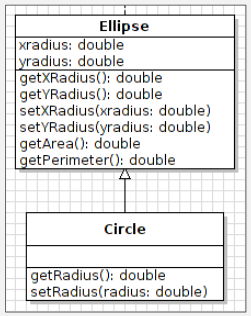
But Circle inherits setXRadius() and setYRadius(), allowing the following –

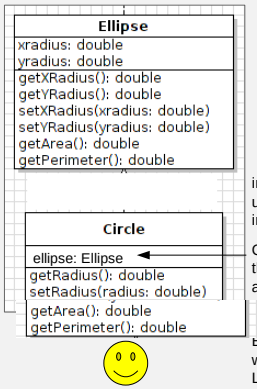
```
Circle c = new Circle();
c.setXRadius(5);
c.setYRadius(10);
```

This doesn't make sense for Circle!
(so no, Circle should not extend Ellipse)

Code Reuse Without Inheritance

- favor composition if LSP does not apply





implement Circle using an Ellipse instance variable

Circle's methods then call the appropriate methods of Ellipse – reuse Ellipse's code without violating LSP

Polymorphism

- an effect of inheritance is that we can avoid repeating code in the *implementation* of different related things
- polymorphism allows us to avoid repeating code in the *usage* of different related things
  - it is legal to write `obj.func(p)` if `obj` is an instance of a type that has a method `func` that takes a parameter of the type `p` is
  - `B extends A` and `B implements A` mean that `B` has all of the method headers that `A` has – so if `obj.func(p)` is legal when `obj` is of type `A`, it is also legal when `obj` is of type `B`

This lets us view a type declaration (for a variable or parameter) as really just a declaration of what operations we might want to use on that object.

- this is the reason behind the advice "declare using the most general type appropriate" – if you don't intend to use a method, don't require that the object support it
  - e.g. use `List<...>` for type declarations and `ArrayList<...>` only for new

Binding

`B extends A` – but what if `B` overrides a method in `A`? Which method body is called?

```
public class A {
  private int x_;
  public A ( int x ) { x_ = x; }
  public int getValue () { return x_; }
}
public class B extends A {
  public B ( int x ) { super(x); }
  public int getValue () { return 2*super.getValue(); }
}
```

- for type checking at compile time, the *declared type* is used – what the compiler can see
- for method invocation at run time, the *actual type* is used – so it does what you want

Consider the code below. What is printed when `main` is run?

```
class Animal {
    void speak() {
        System.out.println("Animal speaks");
    }
}

class Dog extends Animal {
    void speak() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    void speak() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Dog();
        a.speak();
    }
}
```

Animal speaks		0 %
Dog barks	9 respondents	**100 %** ✓
Cat meows		0 %
nothing, this has a compiler error		0 %
nothing, this has a runtime error		0 %