2023-03-13

Unit 9 Review

9.1: Inheritance, Superclass, Subclass

OOP & Inheritance

- One of the most useful features of Object-Oriented programming languages (C++, C#, Java, JavaScript, Kotlin, Python, Ruby, Scala, Swift, ActionScript) is **Inheritance**
- **Inheritance** allows your program to efficiently share common code between different objects **(code reuse)**; helps you better organize your program in ways that model the real world; and create smaller units of maintenance and testing.

OOP & Inheritance

- One of the most useful features of Object-Oriented programming languages (C++, C#, Java, JavaScript, Kotlin, Python, Ruby, Scala, Swift, ActionScript) is **Inheritance**
- **Inheritance** allows your program to efficiently share common code between different objects **(code reuse)**; helps you better organize your program in ways that model the real world; and create smaller units of maintenance and testing.

Person
name
address

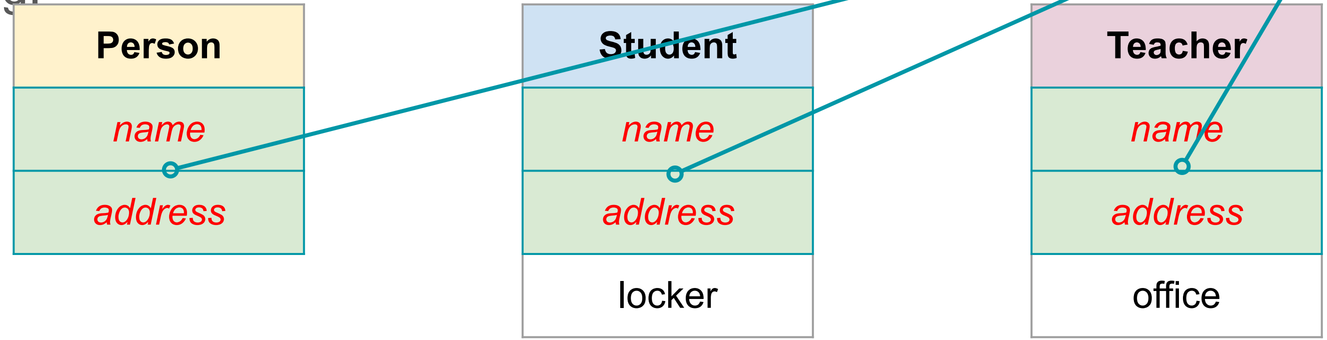
Student
name
address
locker

Teacher
name
address
office

OOP & Inheritance & Generalization

- One of the most useful features of Object-Oriented programm (C++, C#, Java, JavaScript, Kotlin, Python, Ruby, Scala, Swif s **Inheritance**
- **Inheritance** allows your program to efficiently share common different objects **(code reuse)**; helps you better organize your program in ways that model the real world; and create smaller units of maintenance and testing.

Identifying and centralizing common information is called "generalization"



Person	Student	Teacher
name	*name*	*name*
address	*address*	*address*
	locker	office

OOP & Inheritance & Generalization

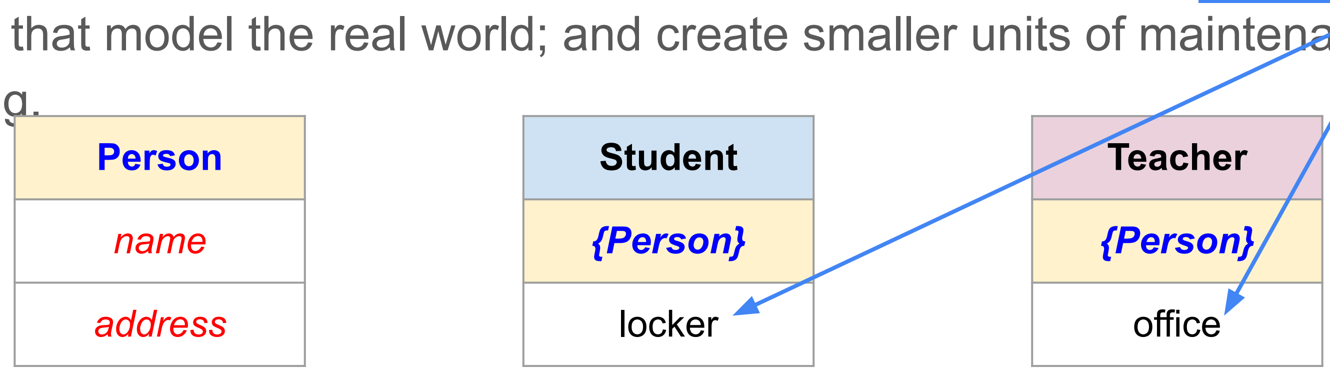
- One of the most useful features of Object-Oriented programming languages (C++, C#, Java, JavaScript, Kotlin, Python, Ruby, Scala, Swift, ActionScript) is **Inheritance**
- **Inheritance** allows your program to efficiently share common code between different objects **(code reuse)**; helps you better organize your program in ways that model the real world; and create smaller units of maintenance and testing.

Person	**Student**	**Teacher**
name	***{Person}***	***{Person}***
address	locker	office

OOP & Inheritance & Generalization & Specialization

- One of the most useful features of Object-Oriented programming languages (C++, C#, Java, JavaScript, Kotlin, Python, Ruby, Scala, Swift
  Inheritance
- **Inheritance** allows your program to efficiently share common different objects **(code reuse)**; helps you better organize your ways that model the real world; and create smaller units of maintenance and testing.

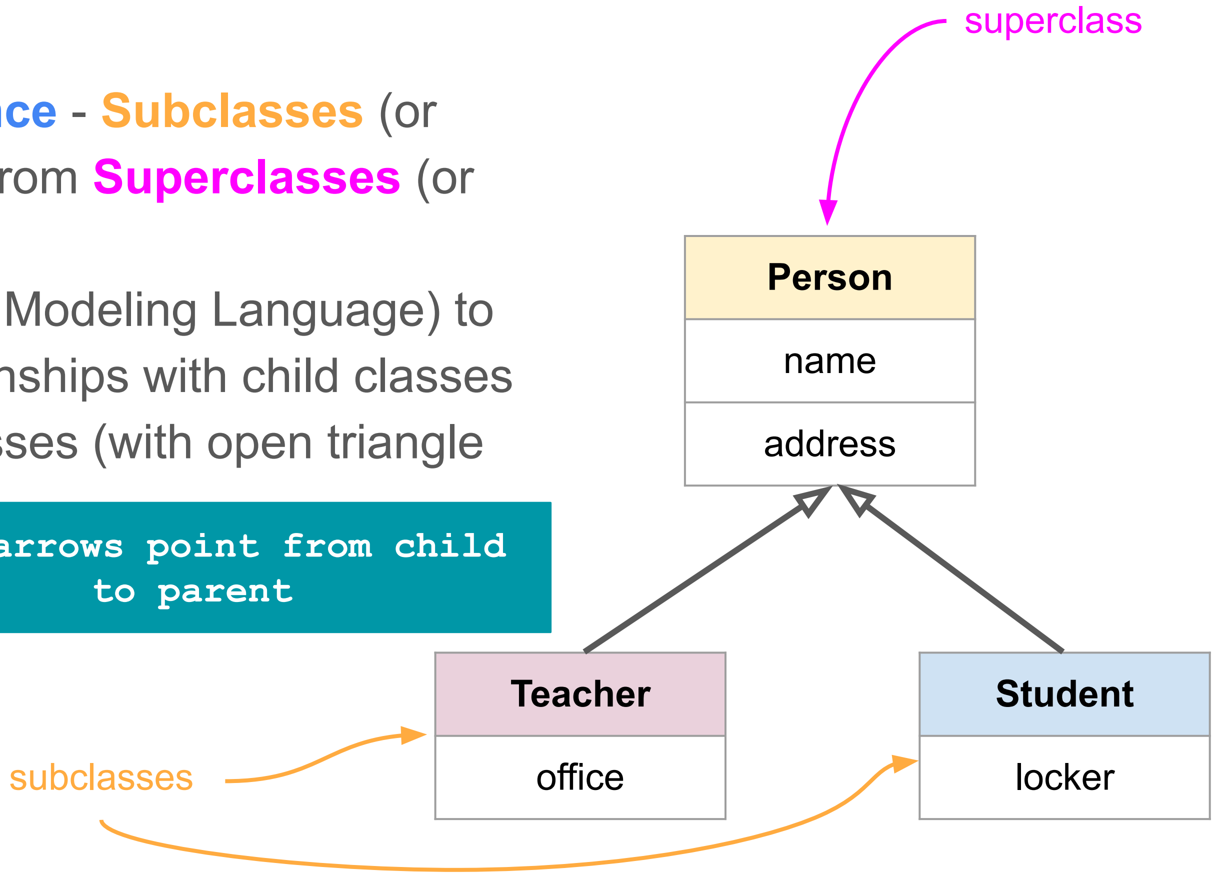
Placing class-specific information in that class is called "specialization"



Superclasses & Subclasses & UML

- When using **Inheritance** - **Subclasses** (or child-classes) inherit from **Superclasses** (or parent-classes)
- We use UML (Unified Modeling Language) to describe these relationships with child classes pointing to parent classes (with open triangle endpoints)

```
Open arrows point from child
to parent
```

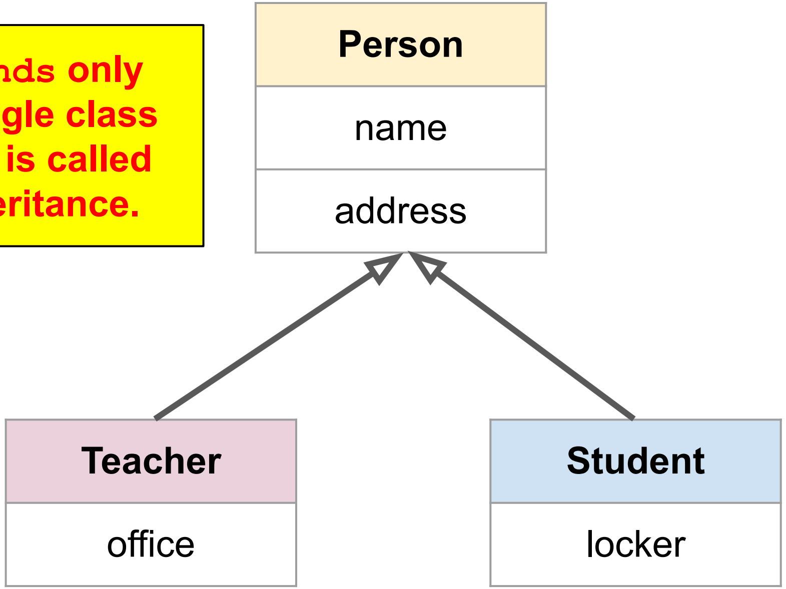


Java & Inheritance

- In Java - any class (**not marked `final`**) can be a superclass - but if a class wants to be a subclass they must use the `extends` keyword

```
class Person {
  public String name;
  public String address;
}

class Teacher extends Person {
  public String office;
}

class Student extends Person {
  public String locker;
}
```

Java extends only allows a single class name - this is called single-inheritance.



Java & Inheritance

Subclasses **inherit** all the variables and methods of their superclass

```
class Person {
  public String name;
  public String address;
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
}

class Student extends Person {
  public String locker;
}
```

```
Person p = new Person();
p.name = "Gary";
p.address = "San Francisco";
p.printInfo();

Teacher t = new Teacher();
t.name = "Chris";
t.address = "San Mateo";
t.printInfo();
t.office = "215W";

Student s = new Student();
s.name = "Beatrice";
s.address = "Colma";
s.printInfo();
s.locker = "B32";
```

Java & Inheritance

Classes that do not use the `extends` keyword automatically extend the `Object` class (has been happening for every class created since Unit 5)

```
class Object {
  public String toString();
  public boolean equals(Object obj);
  ...
}

class Account {
  public String name;
  public double balance;
}
```

```
Object o = new Object();
o.toString();
o.equals(void);


Account a = new Account();
a.toString();
a.equals(void);
a.name = "Amazon";
a.balance = 0.0;
```

Java & Inheritance & `is-a` relationships

Using **Inheritance** results in classes that have `is-a` relationships	
`class Account {}`	`Account is-a Object`

`class Person {}`
`class Teacher extends Person {}`
`class Student extends Person {}`
	`Person is-a Object`
	`Teacher is-a Person / Teacher is-a Object`
	`Student is-a Person / Student is-a Object`

`class Animal {}`
`class Dog extends Animal {}`
`class Snake extends Animal {}`
	`Animal is-a Object`
	`Dog is-a Animal / Dog is-a Object`
	`Snake is-a Animal / Snake is-a Object`

`class Shape {}`
`class Square extends Shape {}`
`class Circle extends Shape {}`
`class Triangle extends Shape {}`
`class Pentagon extends Shape {}`
	`Shape is-a Object`
	`Square is-a Shape / Square is-a Object`
	`Circle is-a Shape / Circle is-a Object`
	`Triangle is-a Shape / Triangle is-a Object`
	`Pentagon is-a Shape / Pentagon is-a Object`

Java & Inheritance & `is-a` relationships

The **`instanceof`** operator in Java can be used to test for `is-a` relationships

`class Account {}`	`Account a = new Account();`
	`System.out.println(a instanceof Object); // true`
	`System.out.println(a instanceof Account); // true`

`class Person {}`
`class Teacher extends Person {}`
	`Person p = new Person();`
	`System.out.println(p instanceof Object); // true`
	`System.out.println(p instanceof Person); // true`
	
	`Teacher t = new Teacher();`
	`System.out.println(t instanceof Object); // true`
	`System.out.println(t instanceof Person); // true`
	`System.out.println(t instanceof Teacher); // true`

Containment & `has-a` relationships

Another concept utilized by Object-Oriented programming languages is **Containment** - where a class is responsible for maintaining an instance of another class inside itself. This results in a `has-a` relationship. We have been using this quite a lot in our examples and projects

```
class Test {
  public String name;
  public double score;
}

class Course {
  public String name;
  public Test tests[10];
}

class Student {
  public String name;
  public Course courses[5];
}
```

```
Test has-a String (name)

Course has-a String (name)
Course has-a Test[] (tests)

Student has-a String (name)
Student has-a Course[] (courses)
```

Modeling `is-a` & `has-a` Relationships

			`is-a` OR `has-a`

`Pet`	`Cat`	`Dog`	**`Dog is-a Pet`**
			`Cat is-a Pet`

`Student`	`Teacher`	`Class`	**`Class has-a Teacher`**
			`Class has-a Student`

`Book`	`Movie`	`Media`	**`Movie is-a Media`**
			`Book is-a Media`

`Circle`	`Shape`	`Square`	**`Circle is-a Shape`**
			`Square is-a Shape`

`Lunch`	`Meal`	`Food`	**`Lunch is-a Meal`**
			`Meal has-a Food`

9.2: Inheritance and Constructors

Java & Inheritance

Note: Not a best practice to make these `public`

Subclasses can only access the **public** variables and **public** methods of their superclass

```
class Person {
  public String name;
  public String address;
  public void printInfo() {
    String info = buildInfoString();
    System.out.println(info);
  }
  private String buildInfoString() {
    return name + " " + address;
  }
}

class Teacher extends Person {
  public String office;
  public String getBuildInfoString() {
    return buildInfoString();  ** ERROR **
  }
}
```

```
Person p = new Person();
p.name = "Gary";
p.address = "San Francisco";
p.printInfo();




Teacher t = new Teacher();
t.name = "Chris";
t.address = "San Mateo";
t.printInfo();
t.buildInfoString(); ** ERROR **
t.office = "215W";
```

Java & Inheritance

Note: This is the better practice - Keep your variables `private` unless you want other code to mess with 'em!

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();
```

Java & Inheritance

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();
```

```
error: constructor Person in class Person cannot be applied to given
types;
class Teacher extends Person {
^
  required: String,String
  found:    no arguments
  reason: actual and formal argument lists differ in length
```

Java & Inheritance

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address
  }
}

class Teacher extends Person {
  public String office;
}
```

Reminder: If you declare any Constructor, Java will no longer automatically create a no-param constructor for you.

In this example, since Person has a Constructor that requires two parameters, there is no way to create a Person with zero parameters.

And this error is telling you that Teacher is malformed because there is no way to properly create its Person superclass.

```
error: constructor Person in class Person cannot be applied to given
types;
class Teacher extends Person {
^
  required: String,String
  found:    no arguments
  reason: actual and formal argument lists differ in length
```

Java & Inheritance

Solution 1: Add a no-param constructor to Person

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();
```

Java & Inheritance

Solution 1: Add a no-param constructor to Person

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person() {
    // empty
  }
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
```

Adding a no-param constructor to Person "fixes" the error - Java now has a way to create a Teacher and its Person superclass.

But did this really fix the issue?

Java & Inheritance

Solution 1: Add a no-param constructor to Person

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person() {
    // empty
  }
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

Adding a no-param constructor to Person "fixes" the error - Java now has a way to create a Teacher and its Person superclass.

But did this really fix the issue?

Q: What is the output of t.printInfo()?
A: null null

Java & Inheritance

Solution 1: Add a no-param constructor to Person

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person() {
    // empty
  }
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

Adding a no-param constructor to Person "fixes" the error - Java now has a way to create a Teacher and its Person superclass.

But did this really fix the issue?

Q: What is the output of t.printInfo()?
A: null null

So let's try something else...

Java & Inheritance

Solution 2: Call the Person constructor from a Teacher constructor using `super()`

But what happens if items in the superclass are not public?

```java
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
  public Teacher() {
    super("<a name>","<an adddress>");
  }
}
```

```java
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

Java & Inheritance

Solution 2: Call the Person constructor from a Teacher constructor using `super()`

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
  public Teacher() {
    super("<a name>","<an adddress>");
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

Subclasses can invoke a constructor in their superclass with super()

1) super() may only be used on the first line of a subclass constructor
2) the params you pass to super() determine which superclass constructor is invoked

Java & Inheritance

Solution 2: Call the Person constructor from a Teacher constructor using `super()`

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
  public Teacher() {
    super("<a name>","<an adddress>");
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

Subclasses can invoke a constructor in their superclass with super()

1) super() may only be used on the first line of a subclass constructor
2) the params you pass to super() determine which superclass constructor is invoked

Q: Now what is the output of t.printInfo()?
A: <a name> <an address>

Java & Inheritance

Solution 2: Call the Person constructor from a Teacher constructor using `super()`

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
  public Teacher() {
    super("<a name>","<an adddress>");
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher();
t.printInfo();
```

Subclasses can invoke a constructor in their superclass with `super()`

1) `super()` may only be used on the first line of a subclass constructor
2) the params you pass to `super()` determine which superclass constructor is invoked

Q: Now what is the output of `t.printInfo()`?
A: `<a name> <an address>`

Better - But probably still not the best solution...

Java & Inheritance

Solution 3: Invoke super() from a 2-param Teacher constructor

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
  public Teacher(String name, String address) {
    super(name, address);
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher("Chris", "Thilgen");
t.printInfo();
```

Java & Inheritance

Solution 3: Invoke super() from a 2-param Teacher constructor

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
  public Teacher(String name, String address) {
    super(name, address);
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher("Chris", "San Mateo");
t.printInfo();
```

```
Q: Now what is the output of t.printInfo()?
A: Chris San Mateo
```

Java & Inheritance

Solution 3: Invoke super() from a 2-param Teacher constructor

But what happens if items in the superclass are not public?

```
class Person {
  private String name;
  private String address;
  public Person(String name, String address) {
    this.name = name;
    this.address = address;
  }
  public void printInfo() {
    System.out.println(name + " " + address);
  }
}

class Teacher extends Person {
  public String office;
  public Teacher(String name, String address) {
    super(name, address);
  }
}
```

```
Person p = new Person("Gary", "San Francisco");
p.printInfo();

Teacher t = new Teacher("Chris", "San Mateo");
t.printInfo();
```

```
Q: Now what is the output of t.printInfo()?
A: Chris San Mateo

Huzzah!
```

Java & Inheritance - Access Modifiers

- We have previously discussed the `private` and `public` keywords (Access Modifiers) and how they are used inside a class to block or allow access to internal methods and variables to code outside the class
- `protected` is another Access Modifier that can be used to allow access to internal methods and variables to **subclasses** of the class (so it is mostly like `private` except for subclasses)

```
class Person {
  public String name;
  protected String age;
  private String taxId;
}

class Teacher extends Person {}

class Pet {}
```

	Can Access	Cannot Access
Person	Person.name Person.age Person.taxId	
Teacher	Person.name Person.age	Person.taxId
Pet	Person.name	Person.age Person.taxId

Java & Inheritance

Summary

- Subclasses do not have access to the private variables and private methods of their superclass (create accessor methods or carefully decorate items with `protected`)
- Subclasses can use the **`super()`** function to invoke superclass constructors
- **`super()`** can only be used on the first line of a subclass constructor (to prevent subclasses from interfering with the creation of the superclass)
- The params passed to **`super()`** determine which constructor in the superclass is invoked
- If you do not add a call to **`super()`** in a subclass constructor - Java will automatically add call to **`super()`** with no params - i.e. it will call the no-param constructor of the superclass (to ensure that the super-class is properly created; because the subclass depends on it)

9.3 Overriding Methods

Overriding Methods

Method Overriding - To implement a new version of a method to replace code that would otherwise have been inherited from a superclass

To override a method from a superclass, implement the method in the subclass.

```
public class Turtle {
  private int x, y;
  public void forward(int z) {
    x += z;
  }
  public int getX() {
    return x;
  }
  public void setX(int x) {
    this.x = x;
  }
}
```

```
public class TurboTurtle extends Turtle {
  public void forward(int z) {
    setX(getX() + 100*z);
  }
}

// QUESTION: Why did we have to use the
// accessor methods in TurboTurtle instead
// of just saying x += 100*z?
```

Confuseth them not: Overloading and Overriding

We learned previously about Method Overloading: defining methods with the same name but different method signatures.

When you **overload** a method, you implement a new method with the same name but different parameters.

Overriding a method is very different than overloading a method.

***Overriding* replaces an inherited method.**

***Overloading* creates a complementary method with the same name.**

Overloading

```
public class MyMath {
  public double sqr(double x) { ... }
  public int sqr(int x) { ... }
}
```

Overriding

```
public class MyBetterMath extends MyMath {
  // I have come up with a FASTER,
  // more efficient method to square
  // doubles!
  @Override
  public double sqr(double x) { ... }
}
```

Failure To Override

To override a method, you must match the method signature exactly. (Parameter names can be different... but the method name, return type, and parameter types must all match.)

If you don't match it up, you may accidentally create an overloaded method! And the compiler will not warn you.

```
class Base {
 public void method() {
   System.out.println("Base.method");
 }
}
class OverrideTest extends Base {
 public void methud() {
   System.out.println("OverrideTest.method");
 }
 public static void main(String args[]) {
   new OverrideTest().method();
 }
}
```

Accidental typo

@Override

Annotations are additional information that can be specified on Java methods, variables and classes. They start with @

@Override is an annotation which tells the compiler that you're trying to override a method... and if your declaration doesn't actually override a method, you get a helpful compile error.

(Why an annotation, not a keyword like C#? Adding keywords to an existing language is hard.)

Trying to override: public boolean equals(Object other); in class Object

```
class Rectangle {
 @Override
 public boolean equals(Rectangle other) {
   return other != null &&
          left == other.left &&
          top == other.top &&
          right == other.right &&
          bottom == other.bottom;
 }
}
```

```
Rectangle.java:3: error: method does not
override or implement a method from a
supertype
 @Override
 ^
1 error
```

The `Object` Superclass

All classes are subclasses of the `Object` class.

(Not necessarily direct subclasses... there are often some superclasses in between.)

When you code, you want to use as specific a type as you can, but this is legal:

```
Object myObject = "Hello, world!";
```

because every String is also an Object.

You also can assign a String to an Object variable, or otherwise use a String in any Object context.

It is always legal and requires no special syntax to convert a reference to a superclass reference:

```
String myString = "Hello, world!";
Object myObject = myString;
```

The `Object` Superclass

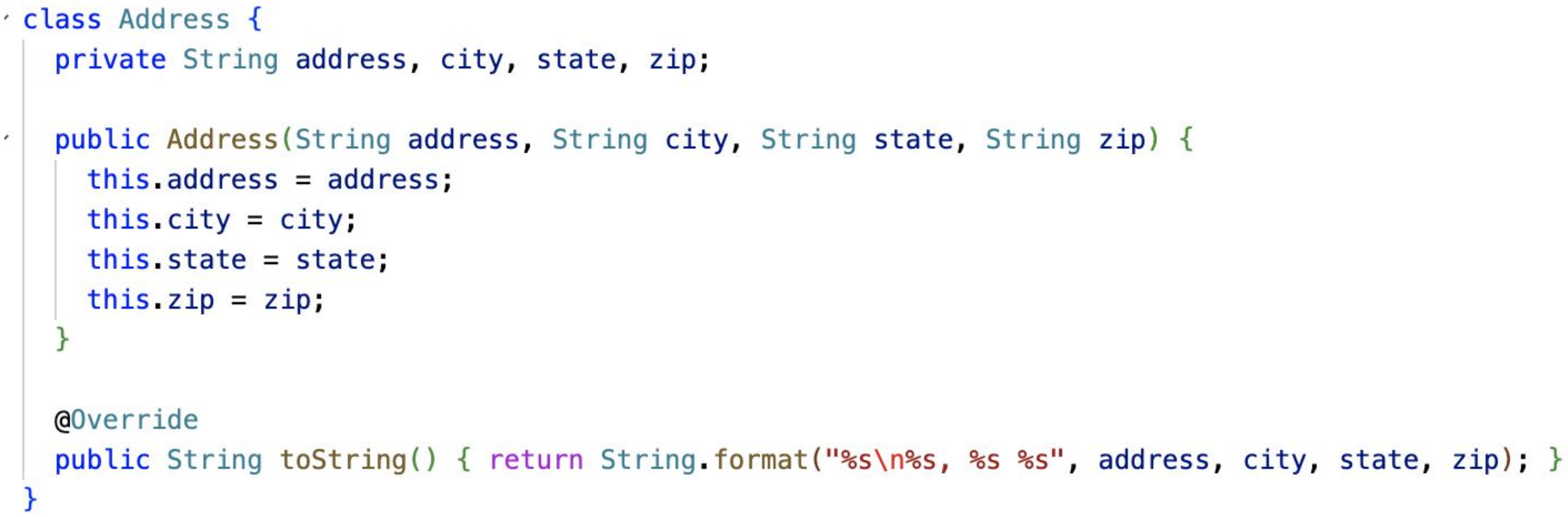
The `Object` class lives in the java.lang package, which is where classes fundamental to the Java language live, like java.lang.String.

`Object` has several public methods. This means that all Java objects inherit these methods. Many of them can be overridden.

Some of the most common ones to override are:

- `public String toString();`
- `public boolean equals(Object o);`

Overriding the `toString` method



```
class Address {
  private String address, city, state, zip;

  public Address(String address, String city, String state, String zip) {
    this.address = address;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  @Override
  public String toString() { return String.format("%s\n%s, %s %s", address, city, state, zip); }
}
```

Overriding the `equals` method

Note that the equals method compares the object with type Object.

So any object can be compared with any other object, of any class, using the equals method!

However, you often want your object to only be equal to objects of the same class, and you may need to compare member variables specific to your class.

What happens if you don't override Object.equals? The default implementation is essentially the same as == on reference types: It returns true if the other object is the exact same object instance.

```
@Override
public boolean equals(Object o)
{
  // returns true or false
}
```

Careful:
public boolean equals(MyClass o) is NOT an override of Object.equals...
It would make a separate overloaded method, with no warning/error unless you say @Override!

Casting

The type cast operator () makes it possible to convert a reference of superclass type (such as Object) to a subclass type (such as Student).

This only works if the superclass reference really IS pointing to an instance of the subclass!

If it isn't, a `ClassCastException` will be thrown.

NOTE: You do NOT need the `(cast)` operator to go from a subclass type to a superclass!

```
Object obj = new String("Hello");
```

```
class Student {
  private String name;
  private String id;

  @Override
  public boolean equals(Object o) {
    if (o == null) {
      return false;
    }
    Student student = (Student)o;
    return name.equals(student.name) &&
           id.equals(student.id);
  }
}
```

Bug: This could throw ClassCastException, if one passes, say, a Date or String to this method.

instanceof operator

The instanceof operator lets you check the "is-a" relationship of an object with a class.

x instanceof T evaluates to true if the object reference x is of type T.

Using instanceof, we can check that the cast to Student is safe before doing it.

An alternative is to try/catch ClassCastException, but that's more expensive.

```
class Student {
  private String name;
  private String id;

  @Override
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Student)) {
      return false;
    }
    Student student = (Student)o;
    return name.equals(student.name) &&
           id.equals(student.id);
  }
}
```

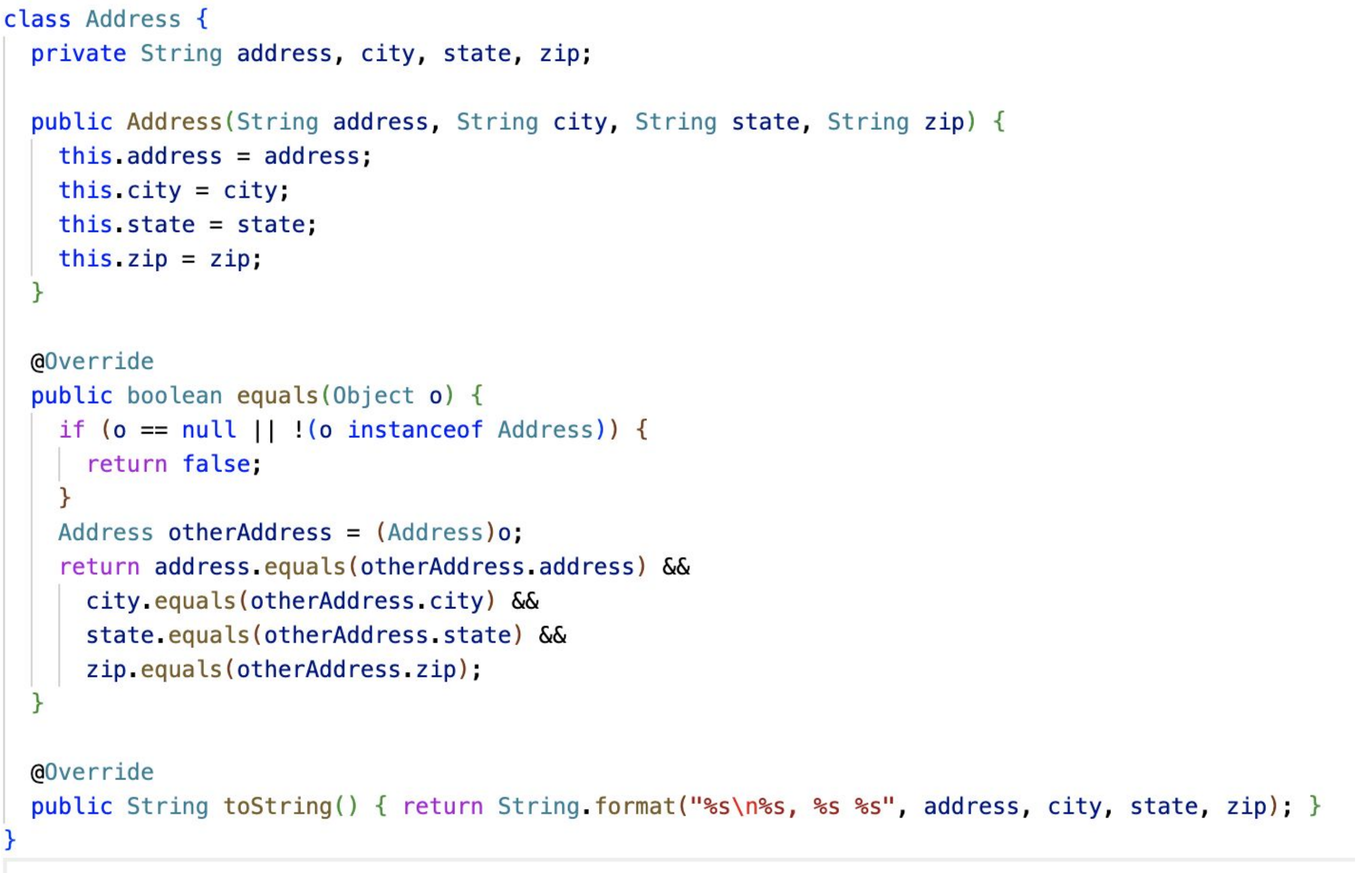
Object.equals contract

Java defines a contract that Object.equals implementations must follow:

Reflexive: x.equals(x) is true
Symmetric: if x.equal(y) is true, y.equals(x) is true
Transient: if x.equals(y) and y.equals(z) are true, x.equals(z) is true
Consistent: x.equals(y) should return the same thing if you call it again, if nothing about them changed
Handles null: x.equals(null) should return false. (And it shouldn't crash with a NullPointerException!)

```
class Student {
  private String name;
  private String id;

  @Override
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Student)) {
      return false;
    }
    Student student = (Student)o;
    return name.equals(student.name) &&
           id.equals(student.id);
  }
}
```

Overriding Object.equals



```
class Address {
  private String address, city, state, zip;

  public Address(String address, String city, String state, String zip) {
    this.address = address;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  @Override
  public boolean equals(Object o) {
    if (o == null || !(o instanceof Address)) {
      return false;
    }
    Address otherAddress = (Address)o;
    return address.equals(otherAddress.address) &&
      city.equals(otherAddress.city) &&
      state.equals(otherAddress.state) &&
      zip.equals(otherAddress.zip);
  }

  @Override
  public String toString() { return String.format("%s\n%s, %s %s", address, city, state, zip); }
}
```

String overrides Object.equals

```
String z = “z”;
String a = z + z;
String b = “zz”;
String c = b;

a == b; // false because a and b refer to different strings
b == c; // true because c and b refer to the same strings

a.equals(b); // true because the values of a and b are the same
c.equals(b); // true because c and b refer to the same string
```

9.4 super.method()

super.method()

We saw super() already in Section 9.2. In that case, super() is used for what’s called “constructor chaining,” where a subclass constructor calls a superclass constructor.

The super keyword can also be used to invoke a superclass's version of a method, even if the subclass overrides the method.

Often, an overridden method wants to do everything the original method did, but add on some additional behavior.

super.method()

```
class Person {
  ...
  public void dump(PrintWriter pw) {
    pw.println("Name: " + name);
  }
}
class Teacher extends Person {
  private String classroom;
  ...
  public void dump(PrintWriter pw) {
    super.dump(pw);
    System.out.println("Classroom: " + classroom);
  }
}
```

super.method() and super() differences

If you don't use super() to do constructor chaining in a subclass, Java essentially does it for you. It will add an implicit super() to call the no-param constructor of the superclass, if one exists.

This is done because Java regards constructors as really important to getting a properly initialized object. You can't skip around a superclass constructor.

Method overrides are different. Java lets you completely replace the definition of a method. The new method code has a choice: It can use super.method() to call the superclass version of the method at some point, or not.

super.method()

```
class NPC {
  ...
  public void tick() {
    // Implements random movement of the NPC around the map
  }
}
class Teacher extends NPC {
  ...
  public void tick() {
    super.tick(); // call super.tick() to get the basic NPC behavior like random movement
    // Additional code here for special NPC behavior specific to this character
  }
}
```

You don't need super.method() all the time!

In a subclass, you can invoke any public or protected method of that class's superclasses.

You don't need to say super.method() to get at these methods. You can just say method().

You only need super.method() when your subclass has overridden method(), but you still need to invoke the original version of the method.

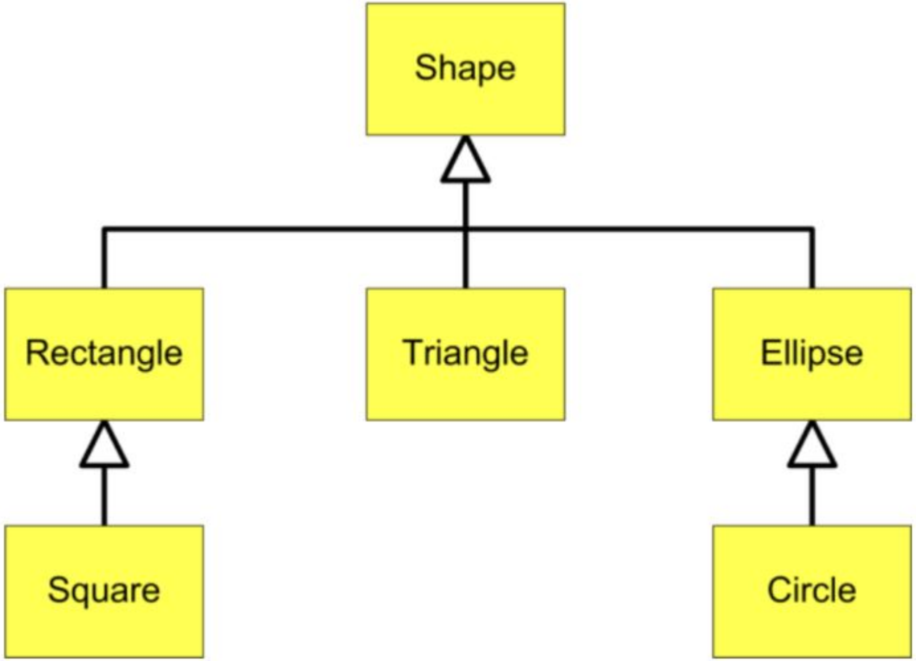
super.method() is usually called from the body of the subclass's implementation of method()!

9.5: Inheritance Hierarchies

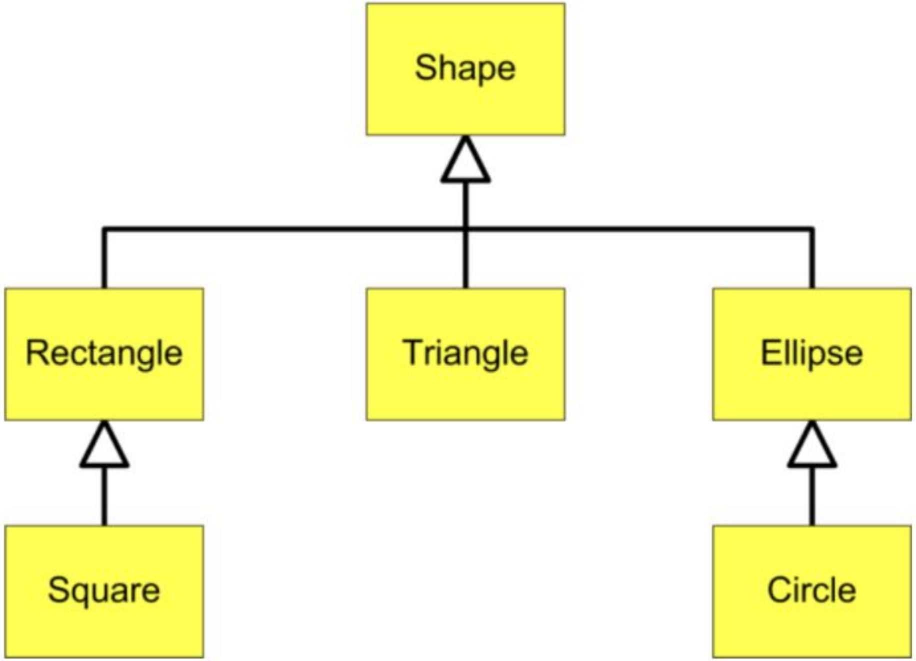
Inheritance Hierarchy

- **Inheritance** allows your program to efficiently share common code between different objects **(code reuse)**; helps you better organize your program in ways that model the real world; and create smaller units of maintenance and testing.
- When you use multiple layers of **Inheritance** in your program you end up with a set of relationships called an Inheritance Hierarchy - most often illustrated as a tree

Geometric Shapes



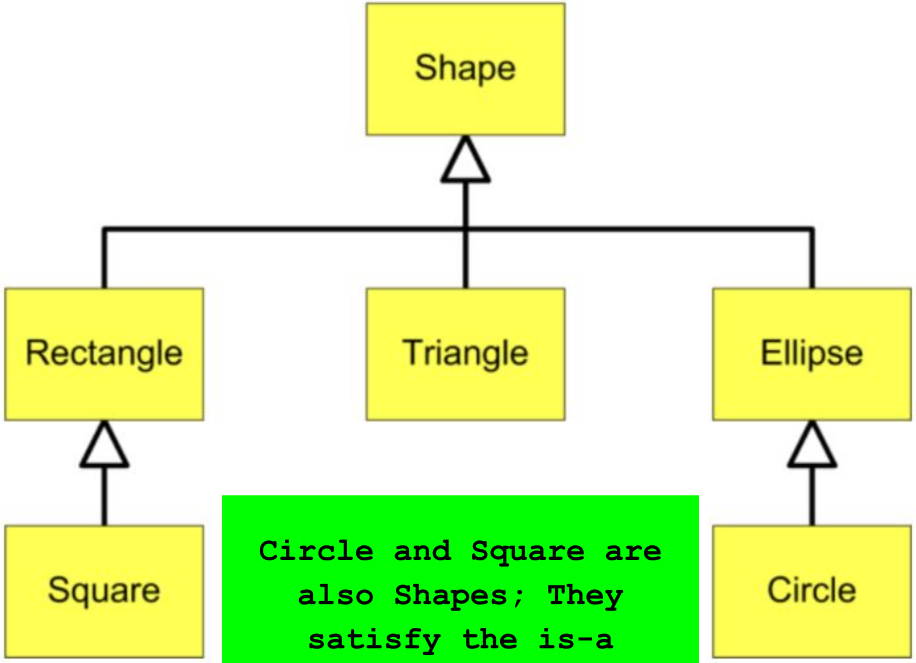
Inheritance Hierarchy



- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

```
Circle is-a Ellipse

Ellipse is-a Shape

Triangle is-a Shape

A Square is-a Rectangle

Rectangle is-a Shape
```

Inheritance Hierarchy

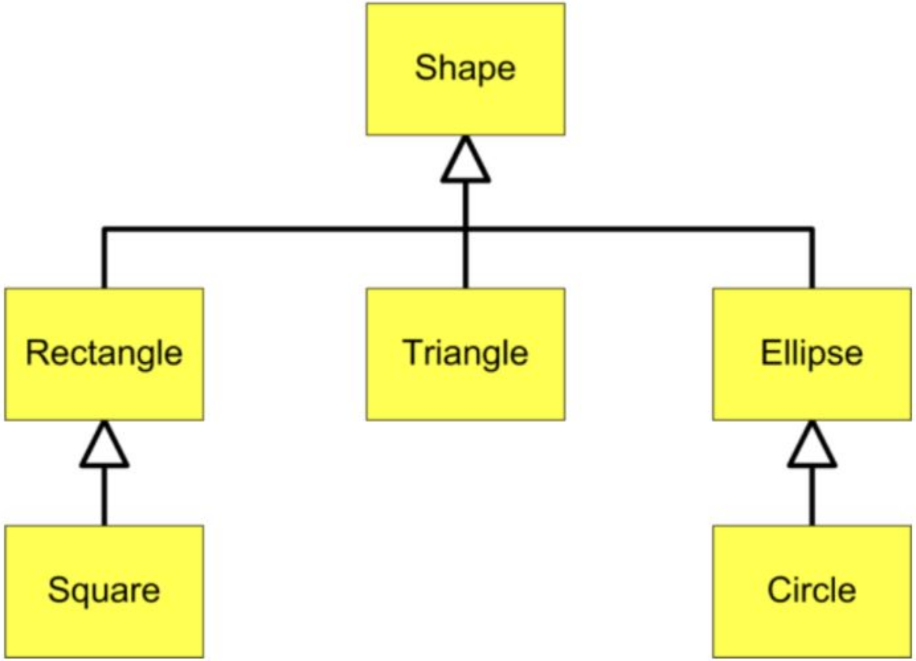


Circle and Square are also Shapes; They satisfy the is-a relationship test with Shape

- This Inheritance Hierarchy shows the relationships between various geometric shapes.
- **Remember:** In UML (Unified Modeling Language) child classes point to parent classes with open triangle endpoints

```
Circle is-a Ellipse

Ellipse is-a Shape

Triangle is-a Shape

A Square is-a Rectangle

Rectangle is-a Shape
```

Inheritance Hierarchy

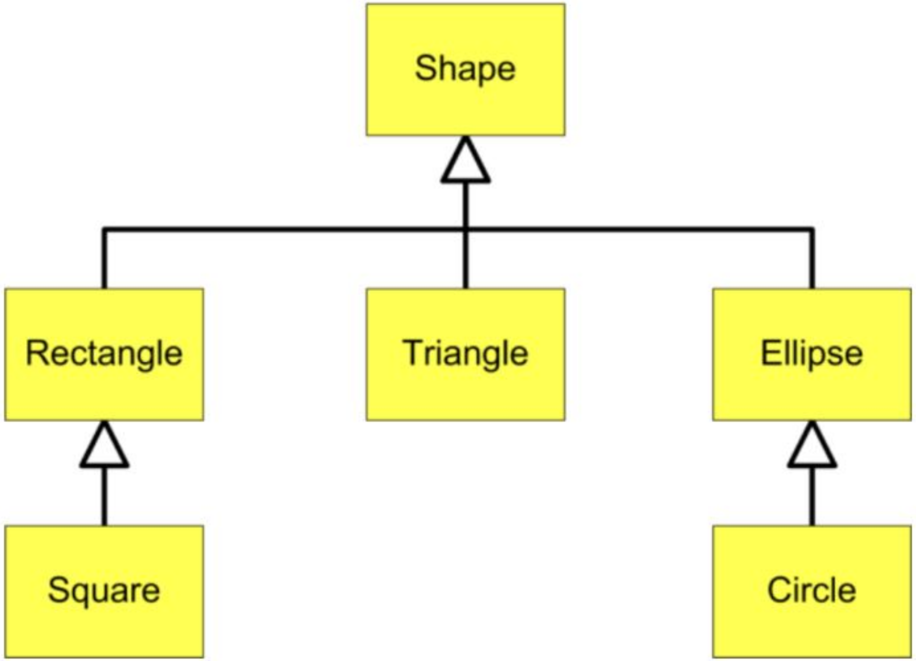


- The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of `Objects`

```
Circle is-a Ellipse
Ellipse is-a Shape
```

```
Circle c = new Circle()
Ellipse e = c;
Shape s = c;
```

Inheritance Hierarchy



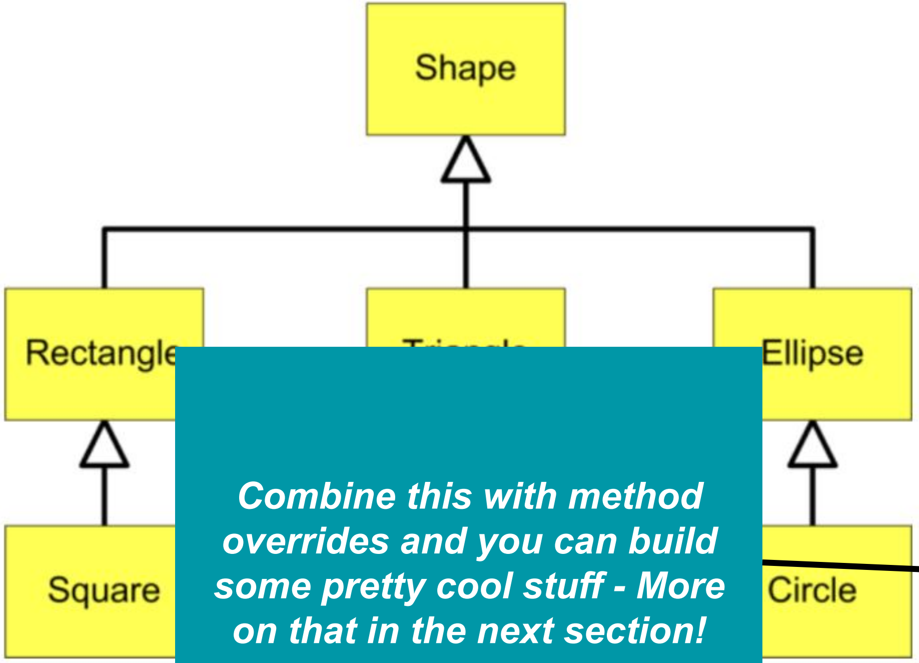
- The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of `Objects`

```
Circle is-a Ellipse
Ellipse is-a Shape
```

```
Circle c = new Circle()
Ellipse e = c;
Shape s = c;
```

This means you can create an `Array/ArrayList` of `Shapes` and add any one of these Objects to it!

Inheritance Hierarchy



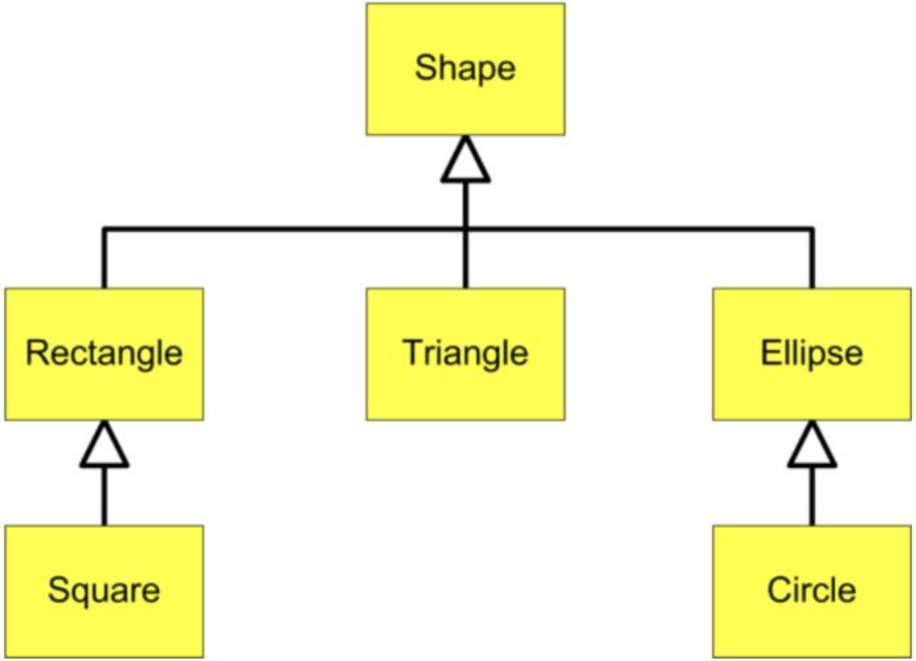
Combine this with method overrides and you can build some pretty cool stuff - More on that in the next section!

- The `is-a` relationship allows you to make use of different types of variable types to hold references to different types of `Objects`

```
Circle is-a Ellipse
Ellipse is-a Shape

Circle c = new Circle()
Ellipse e = c;
Shape s = c;
```

This means you can create an `Array/ArrayList` of `Shapes` and add any one of these Objects to it!

Inheritance Hierarchy

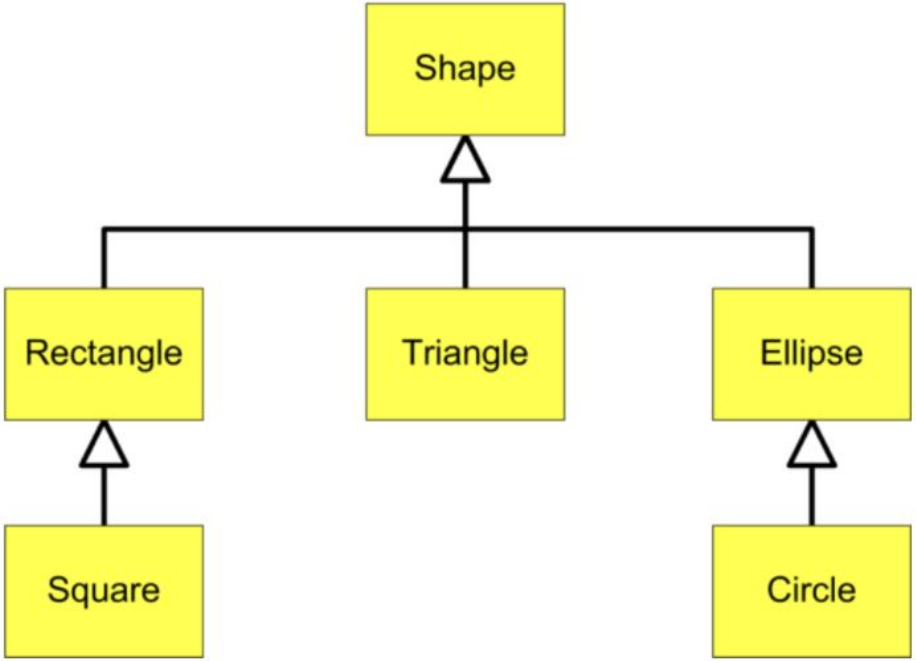


- But this only works in **one direction** - subclass types can become superclass types; but superclass types cannot become subclass types

```
Circle is-a Ellipse
Ellipse is-a Shape
```

```
Shape s = new Shape()
Rectangle r = s;
Triangle t = s;
Ellipse e = e;
```

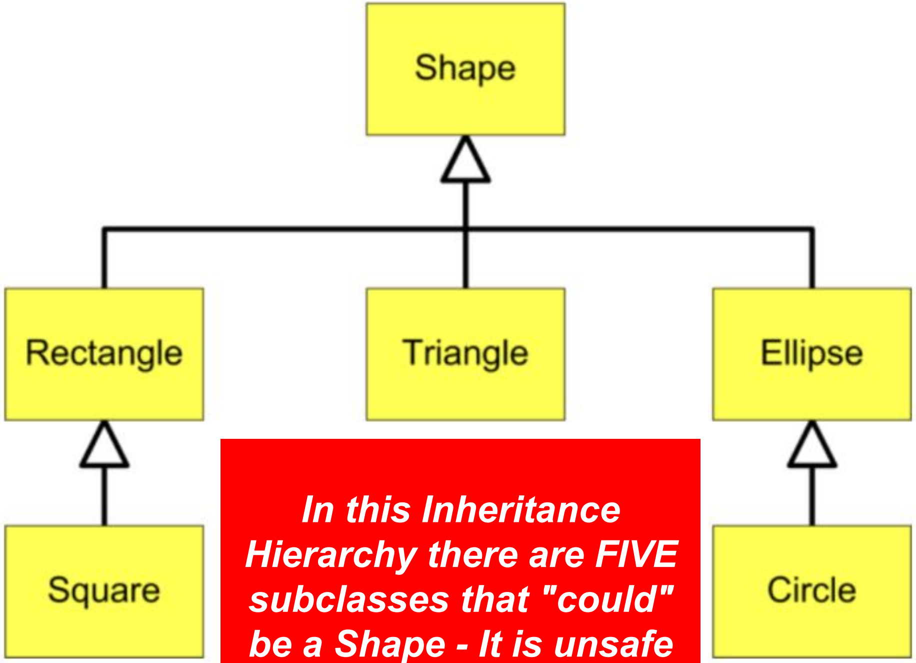
Inheritance Hierarchy



- But this only works in **one direction** - subclass types can become superclass types; but superclass types cannot become subclass types

```
Circle is-a Ellipse
Ellipse is-a Shape

Shape s = new Shape()
Rectangle r = s; **ERROR**
Triangle t = s; **ERROR**
Ellipse e = e; **ERROR**
```

Inheritance Hierarchy



In this Inheritance Hierarchy there are FIVE subclasses that "could" be a Shape - It is unsafe to assume that a Shape is any one of them!

- But this only works in **one direction** - subclass types can become superclass types; but superclass types cannot become subclass types

```
Circle is-a Ellipse
Ellipse is-a Shape

Shape s = new Shape()
Rectangle r = s; **ERROR**
Triangle t = s; **ERROR**
Ellipse e = e; **ERROR**
```

9.6: Polymorphism

Quick review: compile time vs. runtime

- Executing a java program has two steps. First, the program must be **compiled** (e.g. turned into 1s and 0s that your computer can understand) and then **run**
- The first step of this process is orchestrated by a program called a **compiler.**
    - This is the program that yells at you if you try to use a variable before you've initialized it
    - Compilation involves checking a bunch of syntactic "rules" to make sure that your program logic is well-defined
- But you can also encounter error messages generated at **runtime**
    - For example, an error message that says you tried to divide by zero
    - These kinds of error can't be identified a priori–your code needs to be run for these issues to be caught

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

```java
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```java
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}
```

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

```
class Shape {
  public void draw() {
    System.out.println(this.getClass());
  }
}

class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}
```

```
> class Rectangle
> class Triangle
> class Ellipse
```

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

The `Object` instances living in the `shapes` array have compile-time types of `Rectangle`, `Triangle`, and `Ellipse` (because that is the type that was created with `new`)

```
class Rectangle extends Shape {}

class Triangle extends Shape {}

class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}
```

```
> class Rectangle
> class Triangle
> class Ellipse
```

Polymorphism

- In Java when you create an Object with new() - an instance of that specific type is created. The instance will always be an instance of that **compile-time type** regardless of what its current **run-time type** is.

```
class Shape {
  public void draw() {
    System.out.println(this.getClass());
```

As we perform the `for-in` loop - each element of `shapes` is assigned to a `Shape` variable - this is the run-time type of each `Object` instance inside the loop (the compile-time class never changes)

```
class Ellipse extends Shape {}
```

```
Shape shapes[] = new Shape[3];
shapes[0] = new Rectangle();
shapes[1] = new Triangle();
shapes[2] = new Ellipse();

for (Shape s : shapes) {
  s.draw();
}
```

```
> class Rectangle
> class Triangle
> class Ellipse
```

Polymorphism

- The **compiler**
  - Uses the **compile-time type** to verify that the methods you are trying to use are available to an object of that type.
  - The code won’t compile if the methods don’t exist in that class or some parent class of that class.
- During **runtime**
  - Uses the **run-time type** to determine which methods are used
  - When a method is called the first place that is checked for that method is the class that created the object. If the method is found there it will be executed. If not, the parent of that class will be checked and so on until the method is found.

Polymorphism

- **Polymorphic Assignment**

  -
    ```
    Shape s = new Rectangle();
    ```
- **Polymorphic Parameters**

  -
    ```
    public void print(Shape s){}
    ```
- **Polymorphic Collections**

  -
    ```
    Shape[] shapeArray = { new Rectangle(), new Square() };
    ```

Polymorphism

- Polymorphic Assignment
  - `Shape s = new Rectangle();`
- Polymorphic Parameters
  - `public void print(Shape s){}`
- Polymorphic Collections
  - `Shape[] shapeArray = { new Rectangle(), new Square() };`

There are no errors at compile-time because the compiler checks that the “subclass is-a superclass” relationship is true.

At run-time, the Java runtime will use the object’s actual subclass type and call the subclass methods for any overridden methods.

This is why they are polymorphic – the same code can have different results depending on the object’s actual type at run-time.

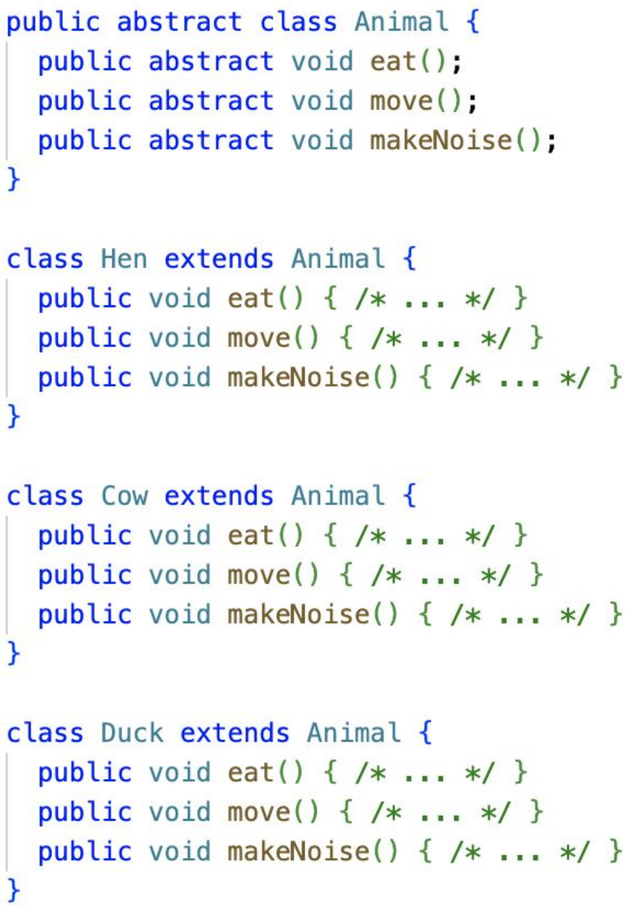
Abstract Classes

Abstract classes are declared with the `abstract` keyword.

Abstract classes cannot be instantiated directly! They must be subclassed.

They usually embody some concept, like `Animal`, that needs to be made *concrete* in a subclass like `Fox`.

Abstract classes are related to polymorphism, in that they enforce the use of polymorphism!



```java
public abstract class Animal {
  public abstract void eat();
  public abstract void move();
  public abstract void makeNoise();
}

class Hen extends Animal {
  public void eat() { /* ... */ }
  public void move() { /* ... */ }
  public void makeNoise() { /* ... */ }
}

class Cow extends Animal {
  public void eat() { /* ... */ }
  public void move() { /* ... */ }
  public void makeNoise() { /* ... */ }
}

class Duck extends Animal {
  public void eat() { /* ... */ }
  public void move() { /* ... */ }
  public void makeNoise() { /* ... */ }
}
```

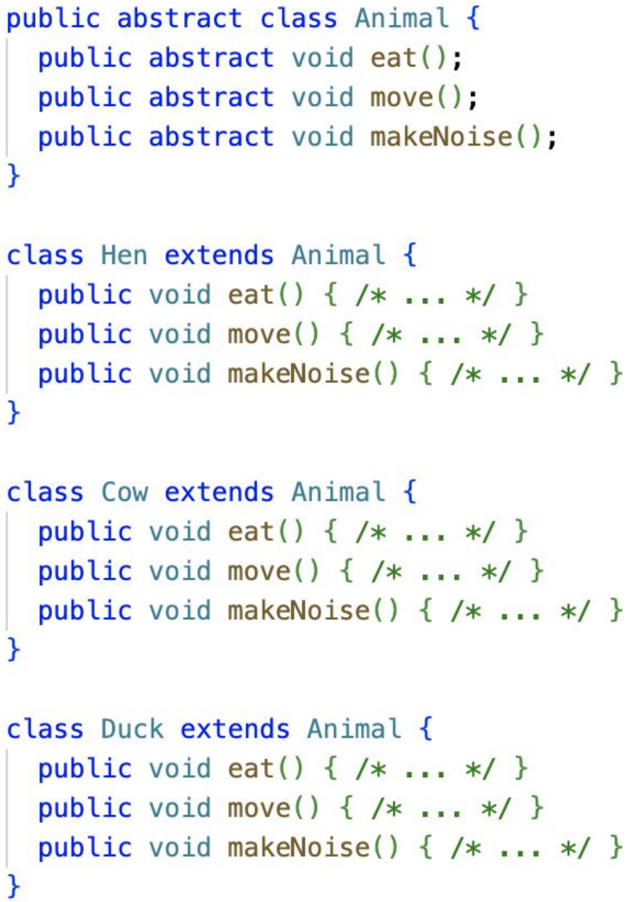
Abstract Methods

Abstract classes often have abstract methods, also defined with the `abstract` keyword.

(A class must be abstract to declare abstract methods. It doesn't have to declare any, though.)

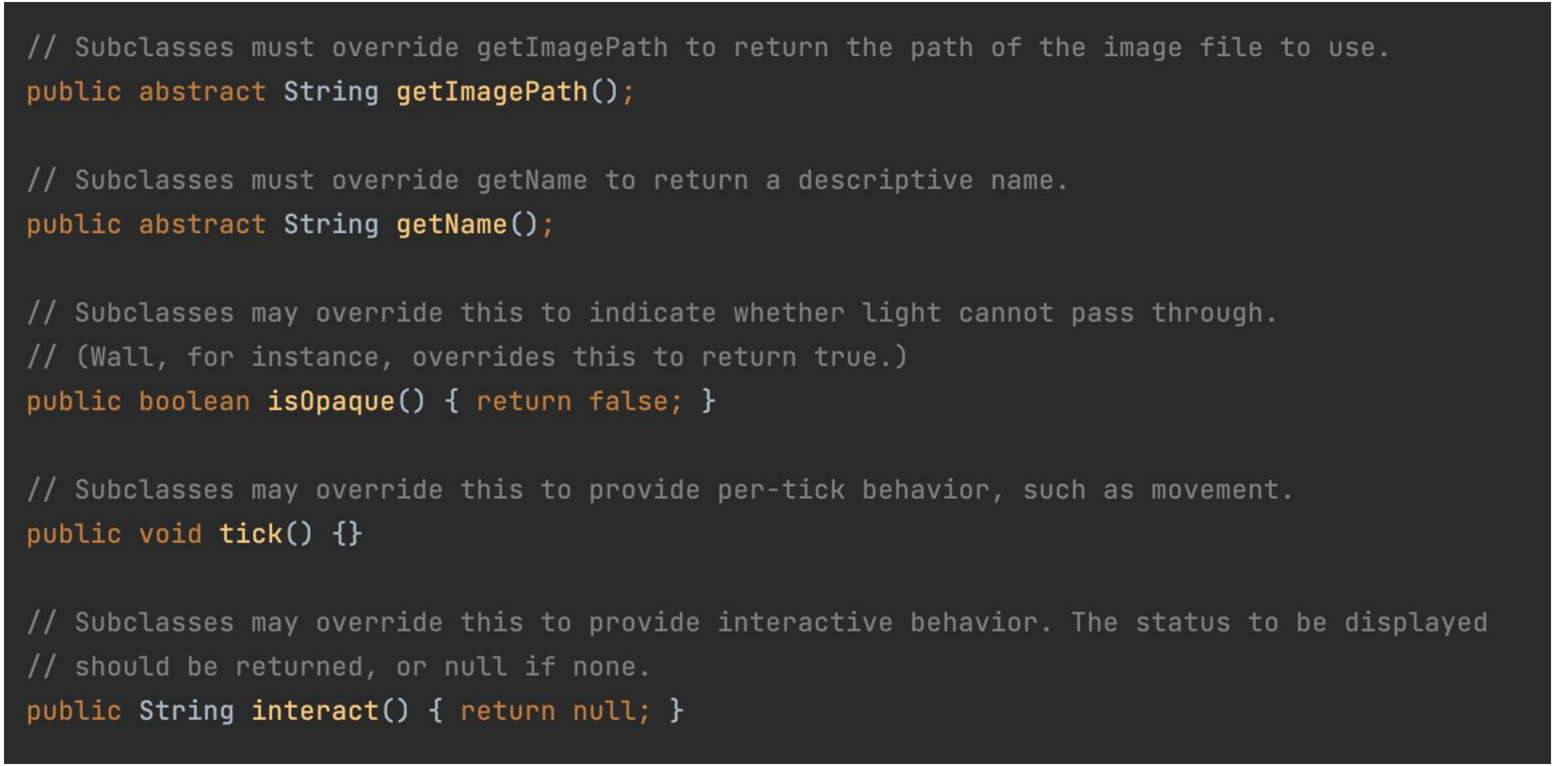
Abstract methods have no method body.

They **must** be overridden by a subclass, or it's a compile error.



```
public abstract class Animal {
  public abstract void eat();
  public abstract void move();
  public abstract void makeNoise();
}

class Hen extends Animal {
  public void eat() { /* ... */ }
  public void move() { /* ... */ }
  public void makeNoise() { /* ... */ }
}

class Cow extends Animal {
  public void eat() { /* ... */ }
  public void move() { /* ... */ }
  public void makeNoise() { /* ... */ }
}

class Duck extends Animal {
  public void eat() { /* ... */ }
  public void move() { /* ... */ }
  public void makeNoise() { /* ... */ }
}
```

MazeObject, an abstract class with abstract methods



```java
// Subclasses must override getImagePath to return the path of the image file to use.
public abstract String getImagePath();

// Subclasses must override getName to return a descriptive name.
public abstract String getName();

// Subclasses may override this to indicate whether light cannot pass through.
// (Wall, for instance, overrides this to return true.)
public boolean isOpaque() { return false; }

// Subclasses may override this to provide per-tick behavior, such as movement.
public void tick() {}

// Subclasses may override this to provide interactive behavior. The status to be displayed
// should be returned, or null if none.
public String interact() { return null; }
```

Interfaces

Languages like Python and C++ support multiple inheritance: Classes can inherit from multiple base classes.

Java has single inheritance. A class can only declare a single superclass.

However, Java has another feature called **interfaces**, which is similar to multiple inheritance.

A class **`extends`** only one superclass, but it **`implements`** zero or more **interfaces**.

Interfaces used to be on the AP Computer Science exam, but were removed in 2017.
So, consider this bonus content.

Interfaces

Interfaces are like abstract classes, except ones where **every** method is abstract.*

An interface is a contract that a class has to implement completely.

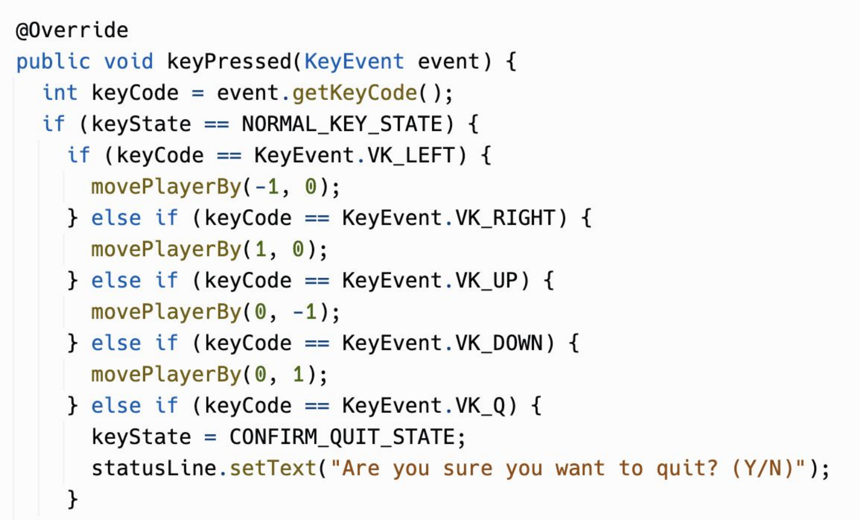
If a class implements KeyListener, it must implement keyPressed, keyTyped and keyReleased, or it's a compile error.

**Mostly true. Default interface methods added in Java 8 (2014)*

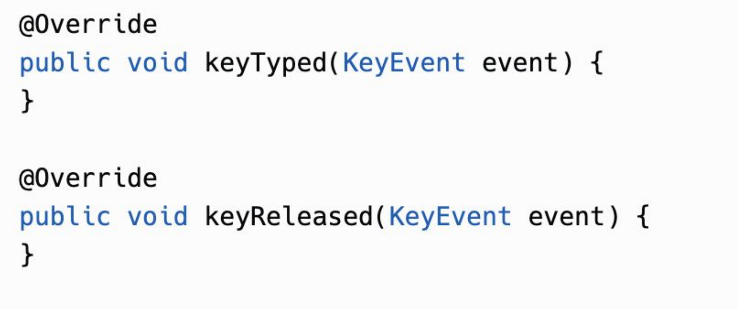
```
/**
 * The listener interface for receiving KeyEvents.
 */
public interface KeyListener extends EventListener {
    /**
     * KEY_PRESSED events are fired when any key (including a function
     * key and cursor key) is pressed while the component has keyboard
     * input focus.
     * KeyEvent.getKeyCode() can be used to find out which key was pressed.
     */
    void keyPressed(KeyEvent ke);

    /**
     * KEY_TYPED events are fired when a key representing a valid text
     * character (not a function key or cursor key) is pressed.
     * KeyEvent.getKeyChar() can be used to get the ASCII code of the key
     * that was pressed.
     */
    void keyTyped(KeyEvent ke);

    /**
     * KEY_RELEASED events are fired when a key is released.
     */
    void keyReleased(KeyEvent ke);
}
```

Game's KeyListener methods

Because Game implements KeyListener, it must implement keyPressed, keyTyped, and keyReleased methods. We only cared about keyPressed, but we had to supply something for the other two.



```java
@Override
public void keyPressed(KeyEvent event) {
  int keyCode = event.getKeyCode();
  if (keyState == NORMAL_KEY_STATE) {
    if (keyCode == KeyEvent.VK_LEFT) {
      movePlayerBy(-1, 0);
    } else if (keyCode == KeyEvent.VK_RIGHT) {
      movePlayerBy(1, 0);
    } else if (keyCode == KeyEvent.VK_UP) {
      movePlayerBy(0, -1);
    } else if (keyCode == KeyEvent.VK_DOWN) {
      movePlayerBy(0, 1);
    } else if (keyCode == KeyEvent.VK_Q) {
      keyState = CONFIRM_QUIT_STATE;
      statusLine.setText("Are you sure you want to quit? (Y/N)");
    }
```



```java
@Override
public void keyTyped(KeyEvent event) {
}

@Override
public void keyReleased(KeyEvent event) {
}
```

- There is a fancy-pants way to avoid the empty methods, an abstract class called KeyAdapter + anonymous inner classes.
- Java 8 (2014) did add "default interface methods" (which have bodies)

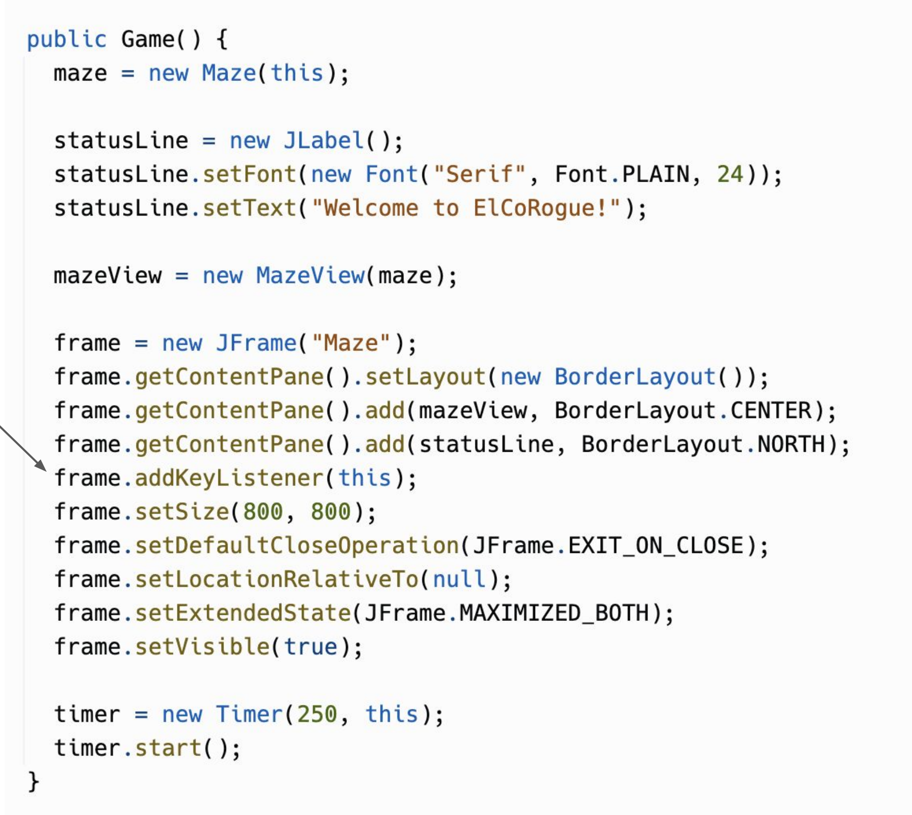
Game's call to addKeyListener

Game registers itself with Java Swing as a key listener by calling addKeyListener on the JFrame that is the game's main window.

The addKeyListener method takes a parameter of type KeyListener. Interfaces are types!

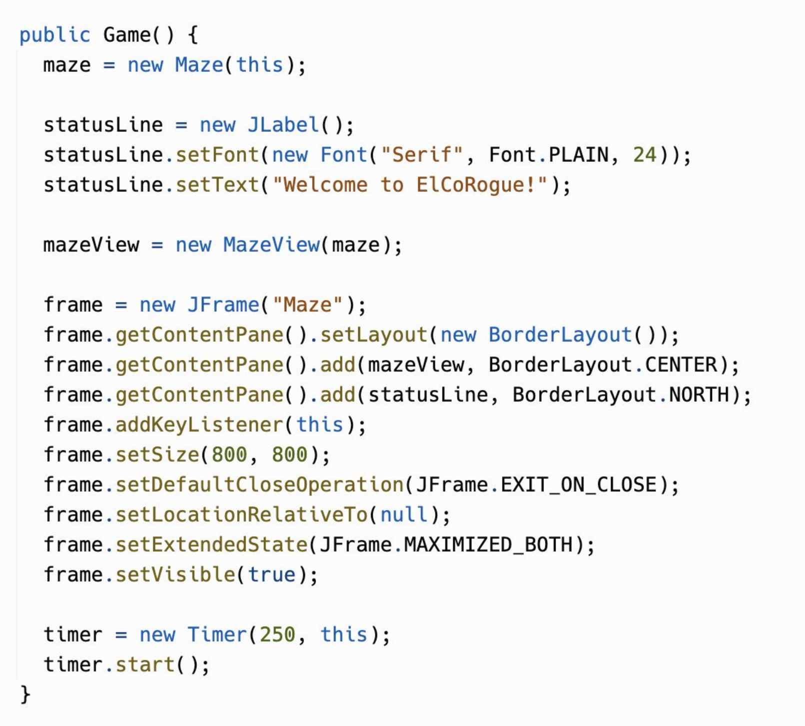
Because Game implements KeyListener, it can be cast to KeyListener.

```
// Legal.
KeyListener k = (KeyListener)game;
```

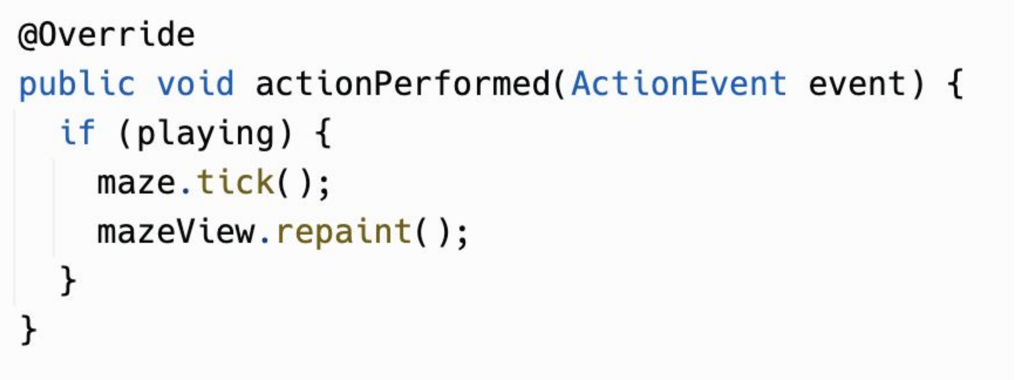


```
public Game() {
  maze = new Maze(this);

  statusLine = new JLabel();
  statusLine.setFont(new Font("Serif", Font.PLAIN, 24));
  statusLine.setText("Welcome to ElCoRogue!");

  mazeView = new MazeView(maze);

  frame = new JFrame("Maze");
  frame.getContentPane().setLayout(new BorderLayout());
  frame.getContentPane().add(mazeView, BorderLayout.CENTER);
  frame.getContentPane().add(statusLine, BorderLayout.NORTH);
  frame.addKeyListener(this);
  frame.setSize(800, 800);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setLocationRelativeTo(null);
  frame.setExtendedState(JFrame.MAXIMIZED_BOTH);
  frame.setVisible(true);

  timer = new Timer(250, this);
  timer.start();
}
```

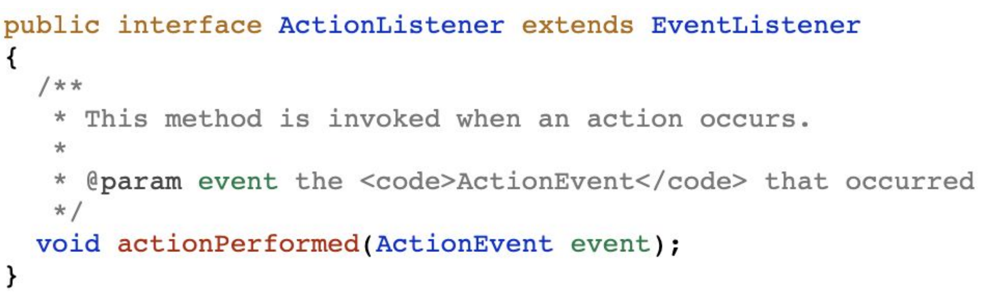
Game implements ActionListener



```java
public Game() {
  maze = new Maze(this);

  statusLine = new JLabel();
  statusLine.setFont(new Font("Serif", Font.PLAIN, 24));
  statusLine.setText("Welcome to ElCoRogue!");

  mazeView = new MazeView(maze);

  frame = new JFrame("Maze");
  frame.getContentPane().setLayout(new BorderLayout());
  frame.getContentPane().add(mazeView, BorderLayout.CENTER);
  frame.getContentPane().add(statusLine, BorderLayout.NORTH);
  frame.addKeyListener(this);
  frame.setSize(800, 800);
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  frame.setLocationRelativeTo(null);
  frame.setExtendedState(JFrame.MAXIMIZED_BOTH);
  frame.setVisible(true);

  timer = new Timer(250, this);
  timer.start();
}
```



```java
@Override
public void actionPerformed(ActionEvent event) {
  if (playing) {
    maze.tick();
    mazeView.repaint();
  }
}
```



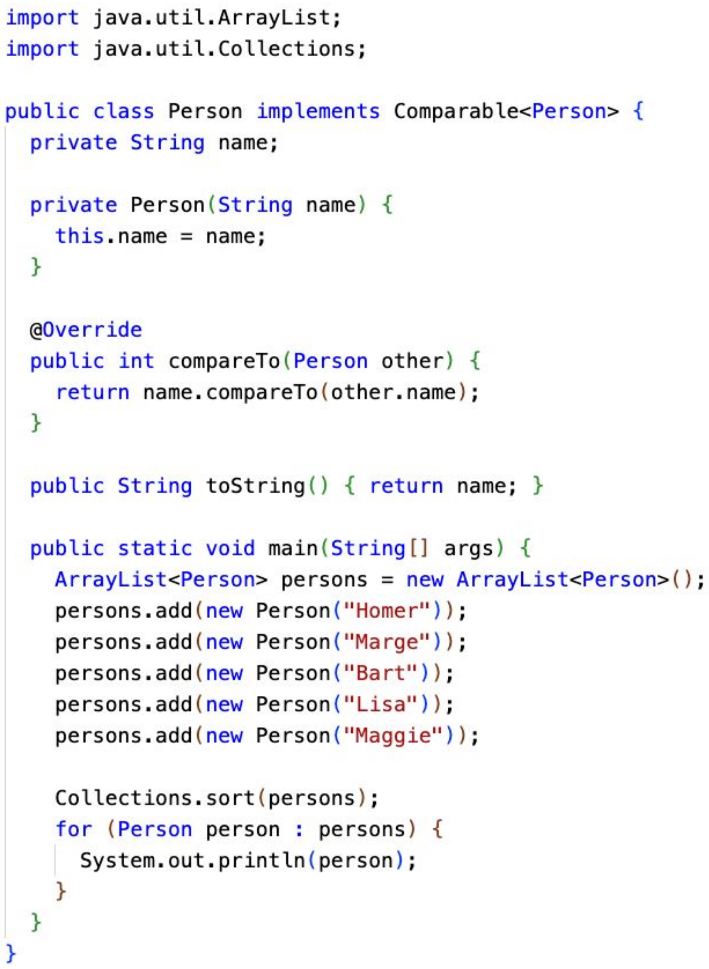
```java
public interface ActionListener extends EventListener
{
  /**
   * This method is invoked when an action occurs.
   *
   * @param event the <code>ActionEvent</code> that occurred
   */
  void actionPerformed(ActionEvent event);
}
```

Comparable<T> interface

Another useful interface to implement is Comparable<T>

This has one method to implement, compareTo(), which should return an integer like String.compareTo()

If your class implements Comparable<T>, a list of your objects can be sorted with Collections.sort()



```java
import java.util.ArrayList;
import java.util.Collections;

public class Person implements Comparable<Person> {
  private String name;

  private Person(String name) {
    this.name = name;
  }

  @Override
  public int compareTo(Person other) {
    return name.compareTo(other.name);
  }

  public String toString() { return name; }

  public static void main(String[] args) {
    ArrayList<Person> persons = new ArrayList<Person>();
    persons.add(new Person("Homer"));
    persons.add(new Person("Marge"));
    persons.add(new Person("Bart"));
    persons.add(new Person("Lisa"));
    persons.add(new Person("Maggie"));

    Collections.sort(persons);
    for (Person person : persons) {
      System.out.println(person);
    }
  }
}
```