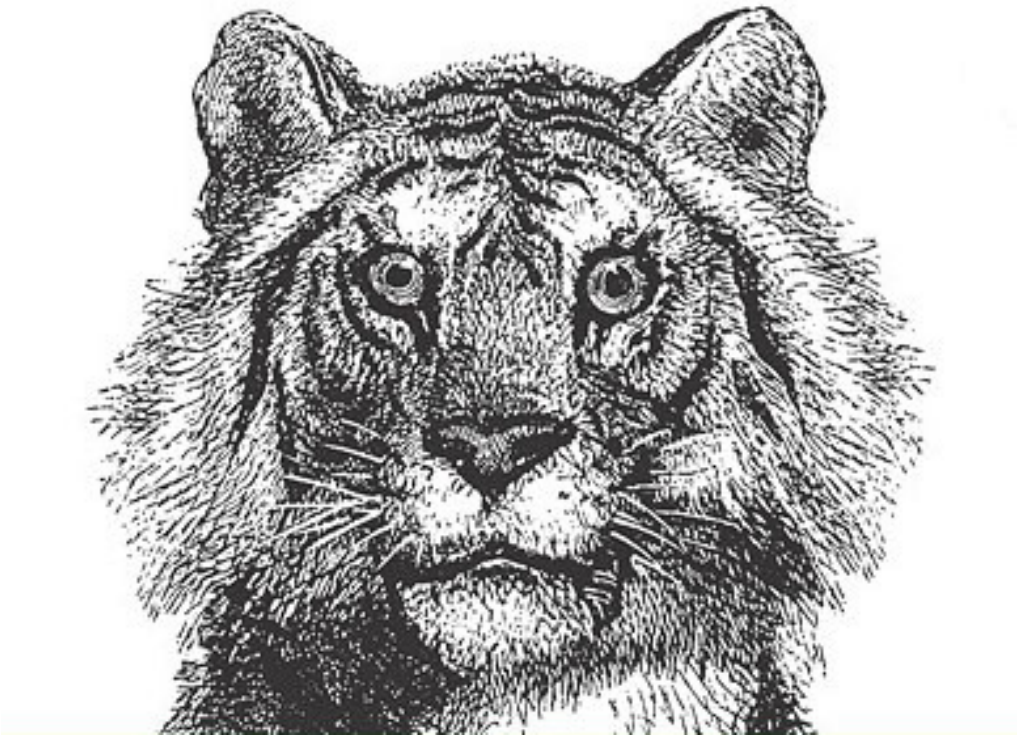
INHERITANCE

Today

- Today we will look into more detail about classes in Java.
- In particular we will look at subclasses and inheritance.
- We'll also talk about interfaces.



- Again, and likely for the last time, lots of this material is drawn from *Java in a Nutshell*, David Flanagan, O'Reilly …

Recap

- Recall the class `Circle` from last lecture.

```
public class Circle{
    private Point location;
    private double radius
}
```

- And recall that it had a sub-class `PCircle`.

```
public class PCircle extends Circle{

}
```

- `PCircle` is a sub-class of `Circle`
- `Circle` is a superclass of `PCircle`.

- Clearly we can define references to both `Circle` and `PCircle`.

```
PCircle pc;
Circle  c;
```

- What is less obvious is that this:

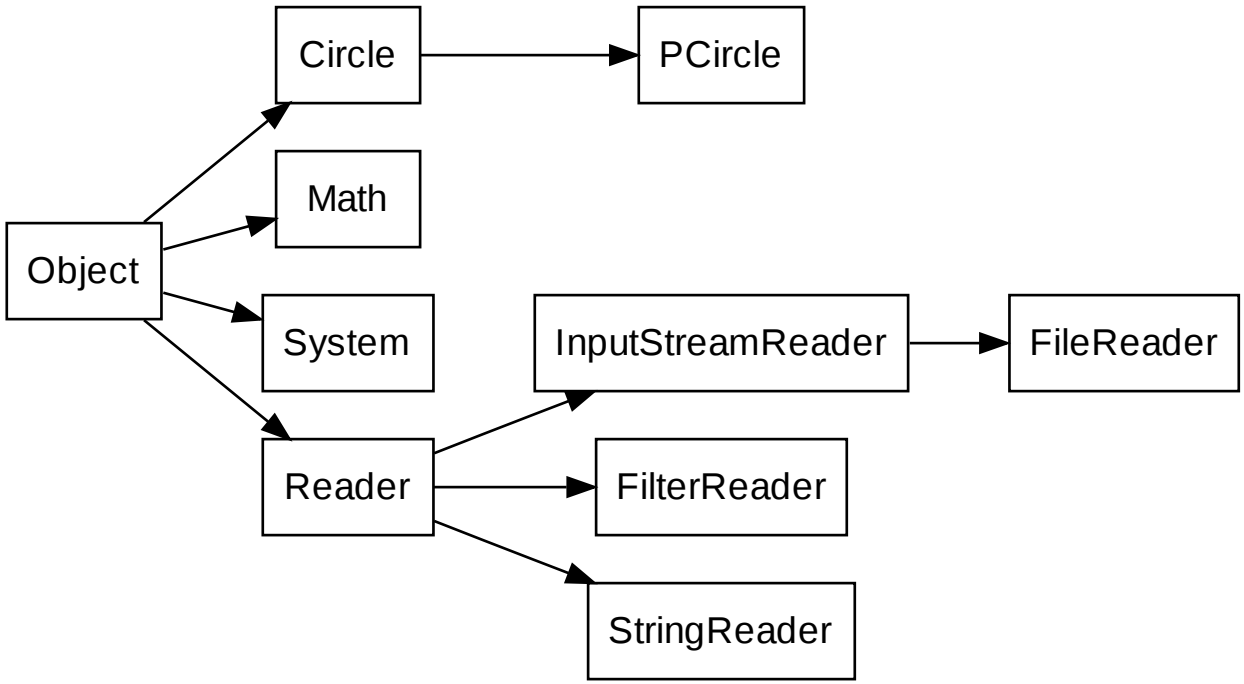
```
c = pc;
```

  is legal without casting.

- Since every `PCircle` is a `Circle`, no conversion is necessary.
- This is one of the neat things that class/subclass relationships allow us to do.

Class Hierarchy

- Superclass and subclass relationships form a *class hierarchy*.
- Every class you define has a superclass.
- If you don't specify a superclass with the `extends` keyword, then `java.lang.Object` is the superclass.
- `java.lang.Object` is the only Java class without a superclass.
- Every object inherits from `java.lang.Object`.


Circle
PCircle
Math
Object
System
InputStreamReader
FileReader
Reader
FilterReader
StringReader

Constructor chaining

- Our definition of `PCircle` included a constructor:

```
public PCircle(){
    super();
    this.origin.setX(0);
    this.origin.setY(0);
}
```

  which explicitly makes a call to the constructor of `Circle`.

- Java makes sure that the constructor for a class is called whenever an object of that class is created.
- Java also makes sure that the constructor is called whenever an object of a subclass is called.
- If the first line of a constructor does not invoke a constructor with:
  - `this();` or
  - `super()`

  Java will insert the call `super()`, the constructor with no arguments.

- When we create a `PCircle`, it will call the constructor above.
- This explicitly calls the constructor for `Circle` using `super()`.
- That call implicitly calls `super()` itself, this time to run the constructor for `Object`.
- The constructor for `Object` runs, then the body of the constructor for `Circle`.
- Finally the constructor for `PCircle` runs.
- Constructor calls are thus *chained*.

- Q: What happens if a class does not have a constructor defined?
- A: Java creates one which is just a call to `super()`, the no argument version of the constructor for the superclass.

Finalizer chaining

- Finalizers do *not* chain.
- If a class finalizer wants to invoke the finalizer of its superclass, it must do so with:

```
super.finalize();
```

Hiding fields

- `Circle` hs a field `radius`.
- Imagine that we add a field `radius` to `PCircle` as well. (This is contrived, but this kind of naming issue does arise).
- How can we refer to the two fields `radius` from within `PCircle` so that we get the one we want?
- `radius` refers to the one in `PCircle`.
- `this.radius` refers to the one in `PCircle`
- `super.radius` refers to the one in `Circle`
- So, `super`, once again, can be used to refer to the superclass.

- You can also refer to `Circle` like this:

  ```
  ((Circle) this).radius
  ```

- Here we cast the `this` reference to be of type `Circle`.
- You can think of this as making `this` refer to the `Circle` part of the class.

- This technique allows us to refer to more than just the superclass.
- Imagine we have classes `A`, `B` and `C`, all with a field `x`.
- `A` is the superclass of `B`, and `B` is the superclass of `C`
- Within C:
  - `x` refers to the field in `C`
  - `this.x` refers to the field in `C`
  - `super.x` refers to the field in `B`
  - `((B) this).x` refers to the field in `B`
  - `((A) this).x` refers to the field in `A`
- Note that `super.super.x` is not legal.

Overriding constructors

- *Hiding* is when a subclass has a field with the same name as a superclass field.
- *Overriding* is when a subclass has a method with the same name as superclass method.
- Unlike hiding, this happens a *lot*
- Typically this is because we want to refine the way the method works to take advantage of the additional fields in the subclass.

A more complex example

- The rest of this lecture uses the Fox/Rabbit example which you can download from the course website.
  - The whole example is in a file `EcoSystem.zip`
- We want a model with several kinds of thing (rabbits and foxes) which have some common features:
  - location in space
  - ability to move
  - color
- So, rather than duplicate code, we create a class hierarchy.

```
public abstract class Animal{
   private Point location;
   protected AnimalColor myColor;

   public setLocationX(int x){
      location.setX(x);
   }
      :

   move(){
      :
   }
```

- Since `Animal` has a `private` field `location`, it provides an API for it.
- This API uses the API for `Point`

  (`Point` is the same class we defined before).
- Here we only show the function to set the x coordinate, but there would be a function to set the y coordinate and functions to get the values of both x and y.
- We might also have a default implementation of the `move` function.

```
public class Rabbit extends Animal{
}

public class Fox extends Animal{
}
```

- Both of these subclasses provide their own implementation of `move()`, overriding the version in `Animal`.
- Note that in Java you cannot call the `move()` method in `Animal` from `Rabbit`.
- If we use:

  ```
  ((Animal) this).move();
  ```

  we will just get the version for `Rabbit`.

- This is a *feature*.
- It is an example of *polymorphism*, the idea that you can use the same code to get different effects depening on what kind of object you pass it.
- For example, you can write code that takes a set of `Animals` as input, and calls `move` on each of them.
- The `Foxes` will move like `Foxes` (using the `Fox move`).
- The `Rabbits` will move like `Rabbits`
- You'll get to do this in the lab and the homework.

Access control

- Access control is to do with the way that fields and methods are called.
- We have seen that `public` fields and methods are accessible from outside the object/class.
- While `private` fields and methods are only accessible from inside the the object/class.
- Java is very strict with the notion of `private`.
- The field `location` in `Animal` is only accessible from within `Animal`.
  - It is not accessible from `Rabbit` or `Fox` without using the API.

- If we want a field or method to be directly accessible from a subclass but we don't want it to be `public`, we can make it `protected`.
- The field `MyColor` in `Animal` is an example.
  - Though I think this would be better as a `private` field, it serves as an example of `protected`.

`final`

- Java places no limits on the length of the class-subclass chain.
  - Given a class, we can always create a subclass of it.

  EXCEPT, when that class was defined using the modifier `final`.
- Thus:

```
public final class lastOne extends previousOne {

}
```

  would prevent anyone from creating a subclass of `lastOne`.
- Many of the Java System classes are final.

- You can also define methods with `final`.
- This prevents them being overridden.
- It also helps to make them run more efficiently since Java doesn't have to check at runtime whether there is another version of the method that needs to be called.
- We have also already seen what using `final` does for a field. (Basically turns the field into a constant).

Abstract classes

- As we have discussed it so far, `Animal` is there to have subclasses made from it.
  - making an instance of `Animal` doesn't make a lot of sense.
- We can formalise this by making it *abstract*:

```
public abstract class Animal{

}
```

- Once it is abstract, it is not possible to make instances of it.

- There is also the concept of an abstract method.
- In our EcoSystem, all the species (sub-classes of `Animal` will define their own way of moving.
  - Nobody will call the version of `move` in `Animal`
- We could just define a version in `Animal` with a blank body, but it is neater to make it abstract:

```
public abstract void move();
```

- Note that an abstract method has no body, just a semicolon.

- Note also that an abstract method can only be defined for a class that is abstract.
- An abstract method is thus like a pure virtual method in C++ in that it forces a class that contains it to be abstract.

Interfaces

- An interface is defined rather like a class:

```
public interface Predator{
void eat();
}
```

- Though it only contains methods (no fields) and all the methods are abstract.
- One important thing about interfaces is that you can create references of that type:

```
Predator p = new Predator();
```

- You can also allow classes to be of the interface type.
  - This gives some of the advantages of multiple-inheritance.

- We associate classes with interfaces by having a class *implement* the interface:

```
public class Rabbit extends Animal
                    implements Predator{


}
```

- With this declaration, `Rabbit` has to provide a definition for all the functions in the `Predator` interface.
  - If it doesn't, the class must be asbtract
- But it now allows us to refer to a `Rabbit` using a `Predator` reference:

```
Rabbit r = new Rabbit();
p = r;
```

- Another way to include polymorphism.

- A class can implement many interfaces.

- Note that there are cases where it is unclear whether something should be an abstract class or an interface.
  - If an abstract class has no data fields and no non-abstract functions, it will look a lot like an interface.
- If this is the case, a common solution is to define *both*.
- First write the interface.
- Then write an abstract class that provides default implementations of some of the methods in the interface.

Summary

- This lecture looked in more detail at subclasses and inheritance. We looked at:
  - hiding,
  - overriding, and
  - polymorphism.
- It also covered some other topics:
  - abstract classes, and
  - interfaces
- This ends the part of the course that it *about* Java specifically.
- Next week we will move on to talk about more general topics and illustrate them using Java.