Classes (Part 3)

SE 206

The return keyword

- The return keyword does a few things:
  - Immediately terminate the current method
- You can have a return anywhere you want
  - Inside loops, ifs, etc.

More on returns

- Consider this class:

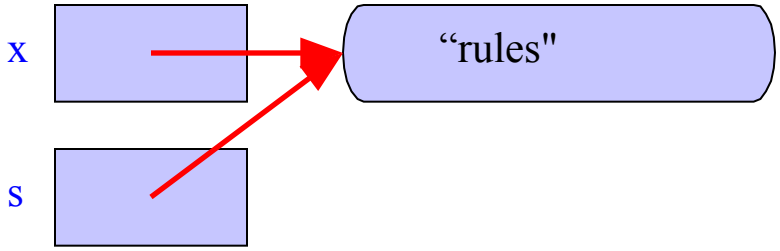
```
public class Foo {
  // Default constructor omitted on this slide
  public String bar (String s) {
    String t = “CS 101” + “ ” + s;
    return t;
  }
}
```

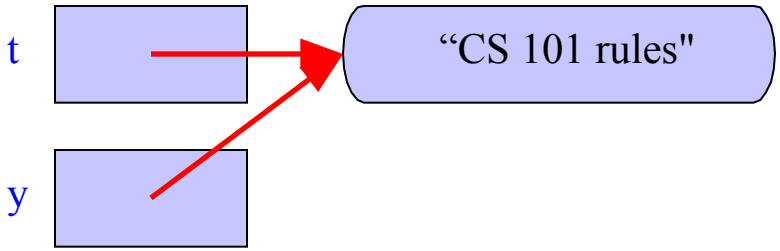
- And the code to invoke it:

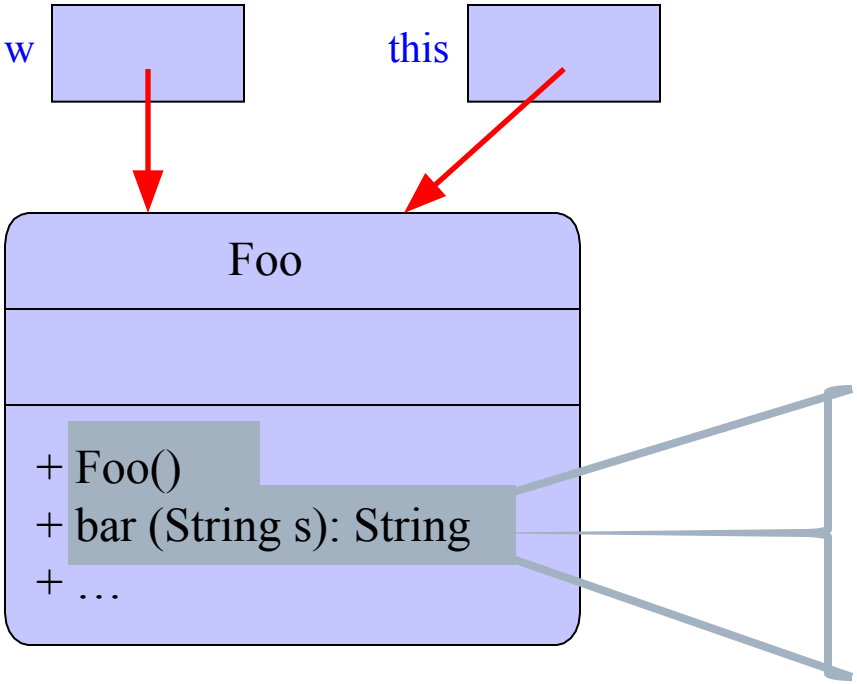
```
Foo w = new Foo();
String x = “rules”;
String y = foo.bar (x);
System.out.println (y);
```

- What happens in memory?

```
Foo w = new Foo();
String x = “rules”;
String y = w.bar (x);
System.out.println (y);
```








```
public String bar (String s) {
  String t = “CS 101” + “ ” + s;
  return t;
}
```

Returning an object from a method

- We could rewrite our bar() method a number of ways:

```
public String bar (String s) {
  String t = “CS 101” + “ ” + s;
  return t;
}

public String bar (String s) {
  return new String (“CS 101” + “ ” + s);
}

public String bar (String s) {
  return “CS 101” + “ ” + s;
}
```

Returning a non-object from a method

- In other words, returning a primitive type from a method

```
public foo () {
  // ...
  return x + y;
}
```

- This method evaluates x+y, then returns that value to the caller

The Circle class

Introducing static-ness, visibilities, etc.

Circle class properties

- What properties does a circle have?
  - Radius
  - PI = 3.14159265358979323 4

Our Circle class

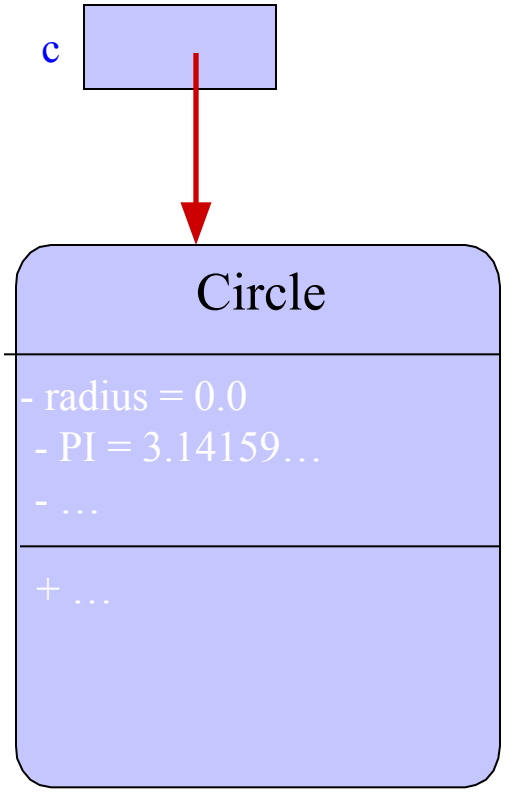
```
Circle c = new Circle();
```

```
public class Circle {
        double radius;
        double PI = 3.1415926536;
}
```

Note the radius field is not initialized by us

We're ignoring the public for now

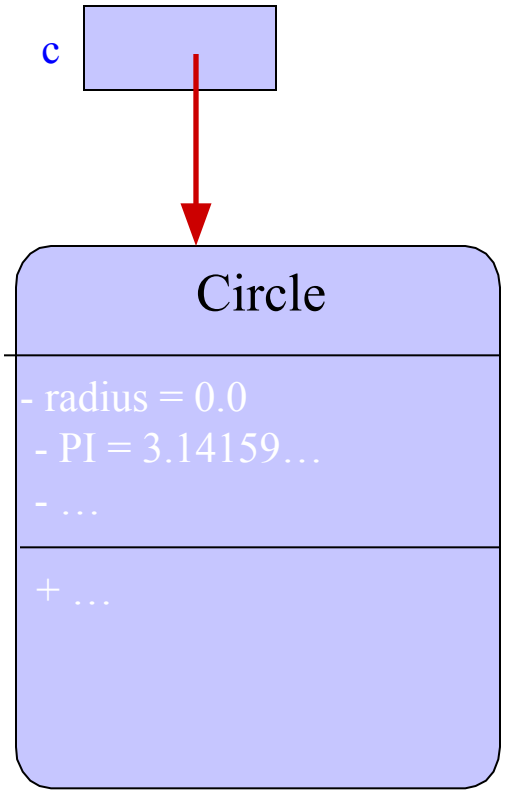
Note the fields are not static



Accessing our Circle object

- Any variable or method in an object can be accessed by using a period
  - The period means 'follow the reference'
  - Example: System.in
  - Example: System.out.println (c.radius);
  - Example: c.PI = 4;

This is bad – PI should have been declared final (this will be done later)



What's the output?

```
public class Circle {
        double radius;
        double PI = 3.1415926536;
}

public class CircleTest {
        public static void main (String[] args) {
                int x;
                Circle c = new Circle();
                System.out.println (x);
        }
}
```

Java will give a "variable not initialized" error

- When a variable is declared as part of a method, Java does *not* initialize it to a default value

What's the output now?

```
public class Circle {
        double radius;
        double PI = 3.1415926536;
}

public class CircleTest {
        public static void main (String[] args) {
                int x;
                Circle c = new Circle();
                System.out.println (c.radius);
        }
}
```

Java outputs 0.0!

- When a variable is declared as part of a class, Java *does* initialize it to a default value

Circle class behaviors

- What do we want to do with (and to) our Circle class?
  - Create circles
  - Modify circles (mutators)
  - Find out about our circles' properties (accessors)
  - Find the area of the circle
  - Plot it on the screen (or printer)
  - A few others…

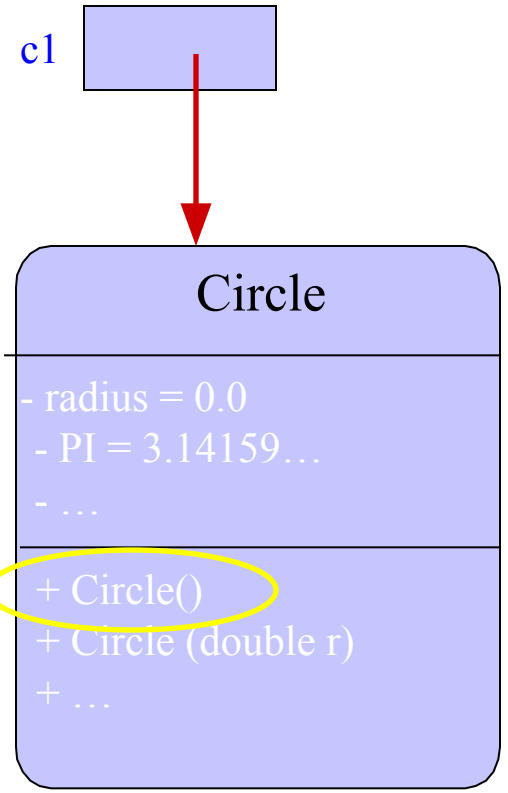
- These will be implemented as methods

Calling the Circle constructor

- To create a Circle object:

  Circle c1 = new Circle();

- This does four things:
  - Creates the c1 reference
  - Creates the Circle object
  - Makes the c1 reference point to the Circle object
  - Calls the constructor with no parameters (the 'default' constructor)



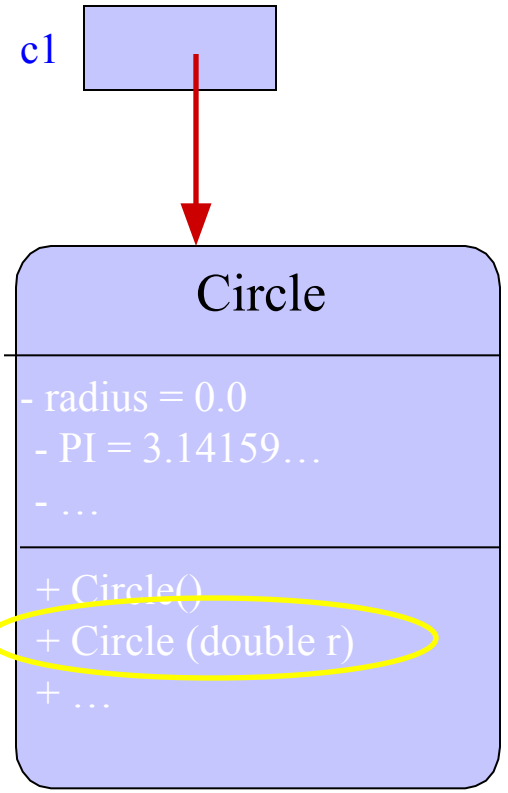
- The constructor is *always* the first method called when creating (or 'constructing') an object

Calling the Circle constructor

- To create a Circle object:

  Circle c1 = new Circle(2.0);

- This does four things:
  - Creates the c1 reference
  - Creates the Circle object
  - Makes the c1 reference point to the Circle object
  - Calls the constructor with 1 double parameters (the 'specific' constructor)



- The constructor is *always* the first method called when creating (or 'constructing') an object

Constructors

- Remember, the purpose of the constructor is to initialize the instance variables
  - PI is already set, so only radius needs setting

```
public Circle() {
        radius = 1.0;
}

public Circle (double r) {
        radius = r;
}
```

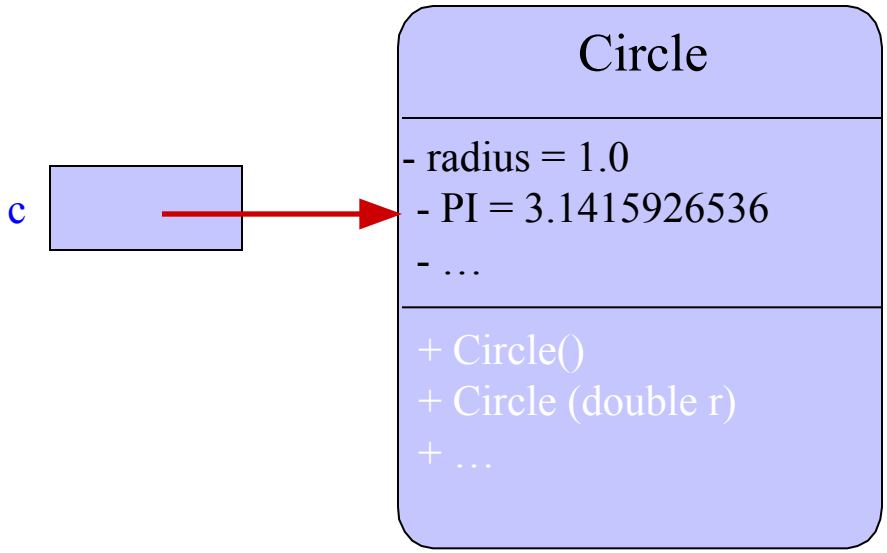
Note there is no return type for constructors

Note that the constructor name is the EXACT same as the class name

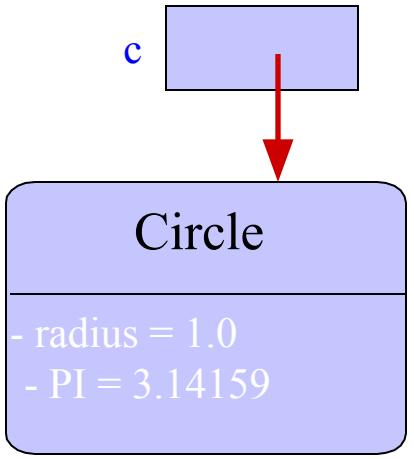
Note that there are two "methods" with the same name!

What happens in memory

- Consider: Circle c = new Circle();
- A double takes up 8 bytes in memory
- Thus, a Circle object takes up 16 bytes of memory
  - As it contains two doubles



Shorthand representation

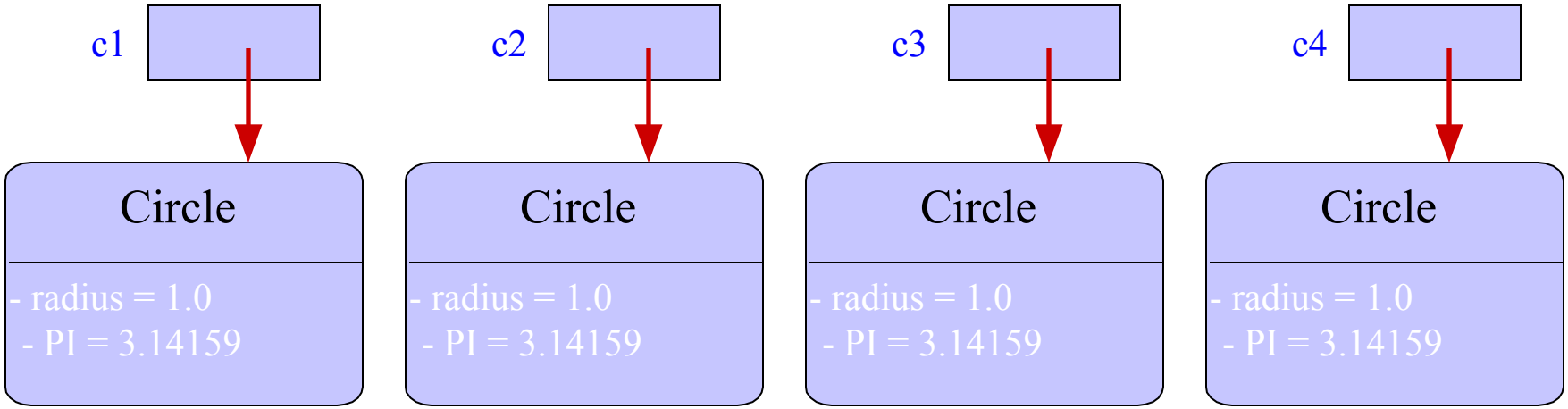


Consider the following code

```
public class CircleTest {
        public static void main (String[] args) {
                Circle c1 = new Circle();
                Circle c2 = new Circle();
                Circle c3 = new Circle();
                Circle c4 = new Circle();
        }
}
```

What happens in memory

- There are 4 Circle objects in memory
  - Taking up a total of 4*16 = 64 bytes of memory



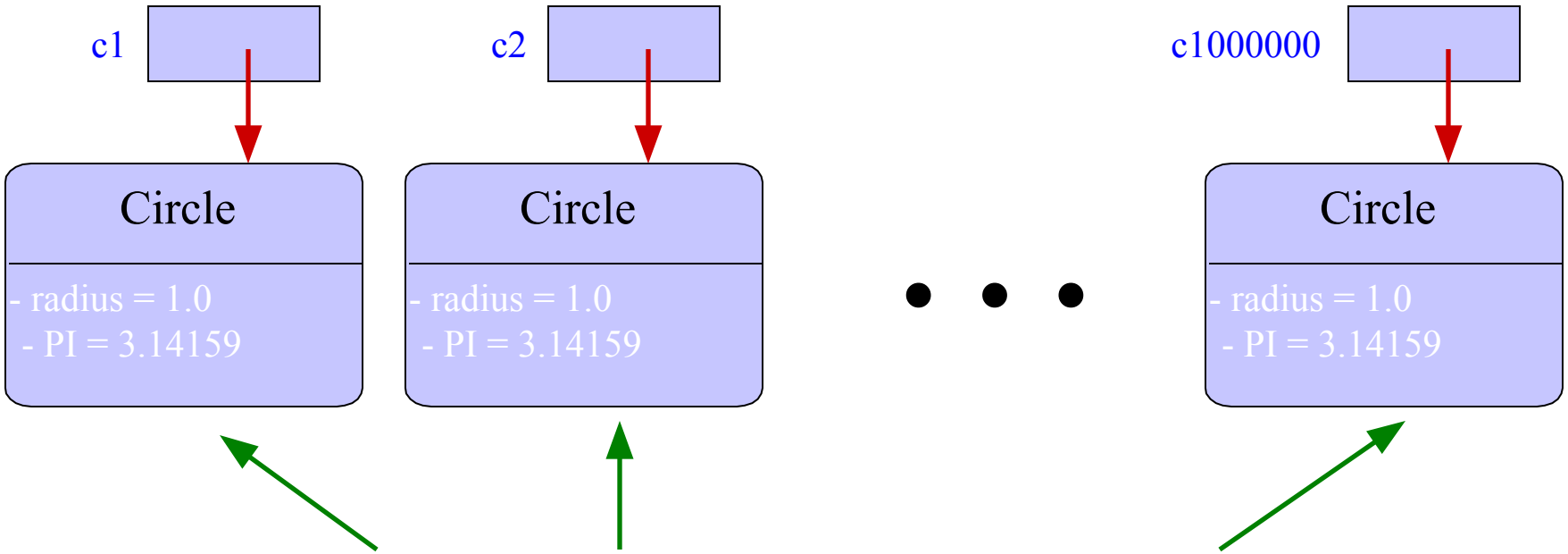
Consider the following code

```
public class CircleTest {
        public static void main (String[] args) {
                Circle c1 = new Circle();
                //...
                Circle c1000000 = new Circle();
        }
}
```

This program creates 1 million Circle objects!

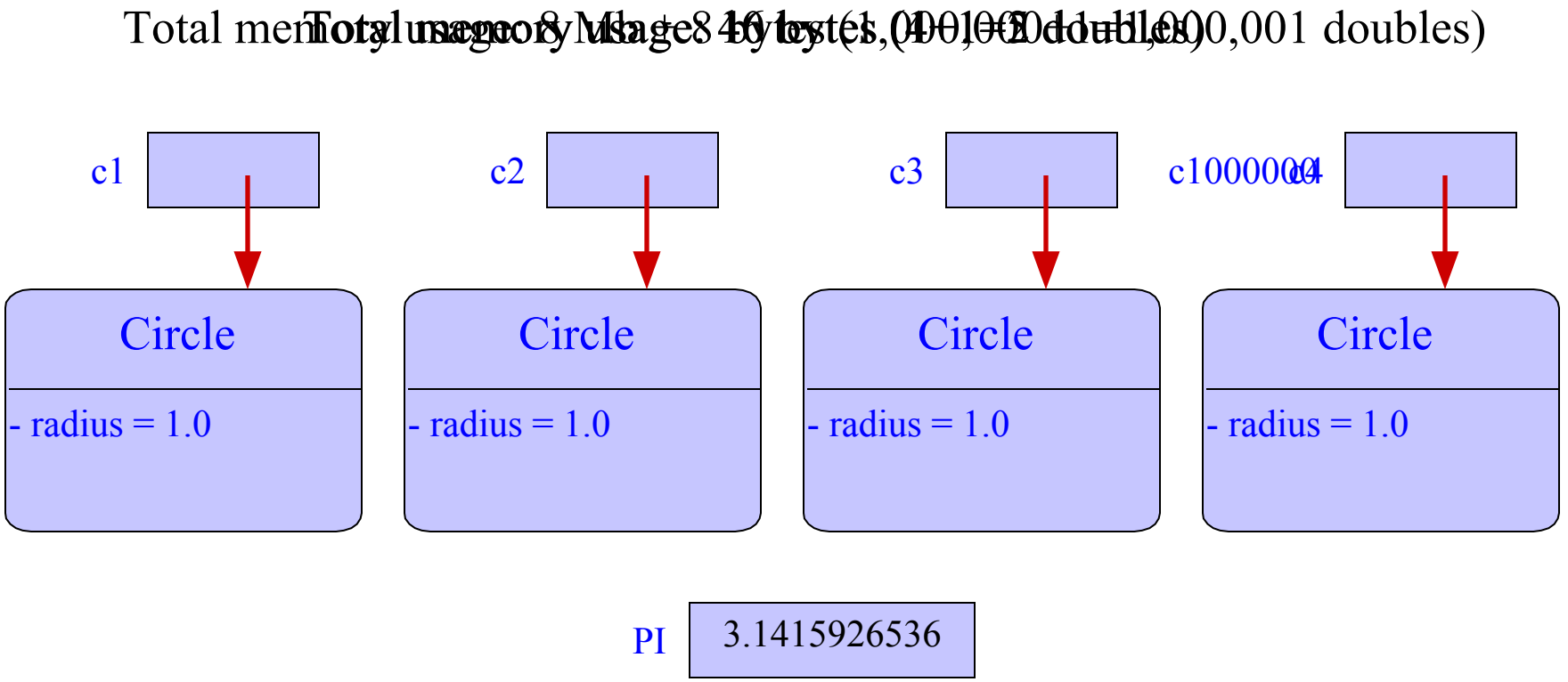
What happens in memory

- There are 1 million Circle objects in memory
  - Taking up a total of 1,000,000*16 ≈ 16 Mb of memory



The use of static for fields

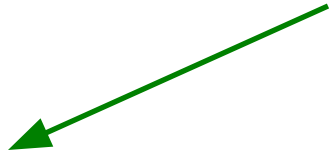
- If a variable is static, then there is only ONE of that variable for ALL the objects
  - That variable is shared by *all* the objects



More on static fields

- What does the following print
  - Note that PI is not final

```
Circle c1 = new Circle();
Circle c2 = new Circle();
Circle c3 = new Circle();
Circle c4 = new Circle();
c1.PI = 4.3;
System.out.println (c2.PI);
```



Note you can refer to static fields by object.variable

- It prints 4.3

Even more on static fields

- There is only one copy of a static field no matter how many objects are declared in memory
  - Even if there are zero objects declared!
  - The one field is "common" to all the objects

- Static variables are called class variables
  - As there is one such variable for all the objects of the class
  - Whereas non-static variables are called instance variables

- Thus, you can refer to a static field by using the class name:
  - Circle.PI

Even even more on static fields

- This program also prints 4.3:

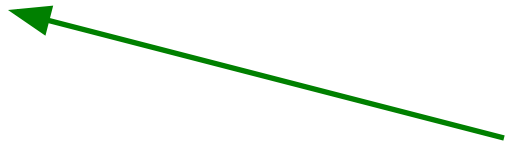
```
Circle c1 = new Circle();
Circle c2 = new Circle();
Circle c3 = new Circle();
Circle c4 = new Circle();
Circle.PI = 4.3;
System.out.println (c2.PI);
```

Even even even more on static fields

- We've seen static fields used with their class names:
  - Math.PI (type: double)
  - Integer.MAX_VALUE (type: int)

Back to our Circle class

```
public class Circle {
        double radius;
        final static double PI = 3.1415926536;

        public Circle() {
                radius = 1.0;
        }

        public Circle (double r) {
                radius = r;
        }

}
```



Note that PI is now final and static

- But it doesn't do much!

Adding a method

```
public class Circle {
        double radius;
        final static double PI = 3.1415926536;

        // Constructors...

        double computeArea () {
      return PI*radius*radius;
   }

}
```

Note that a (non-static) method can use both instance and class variables

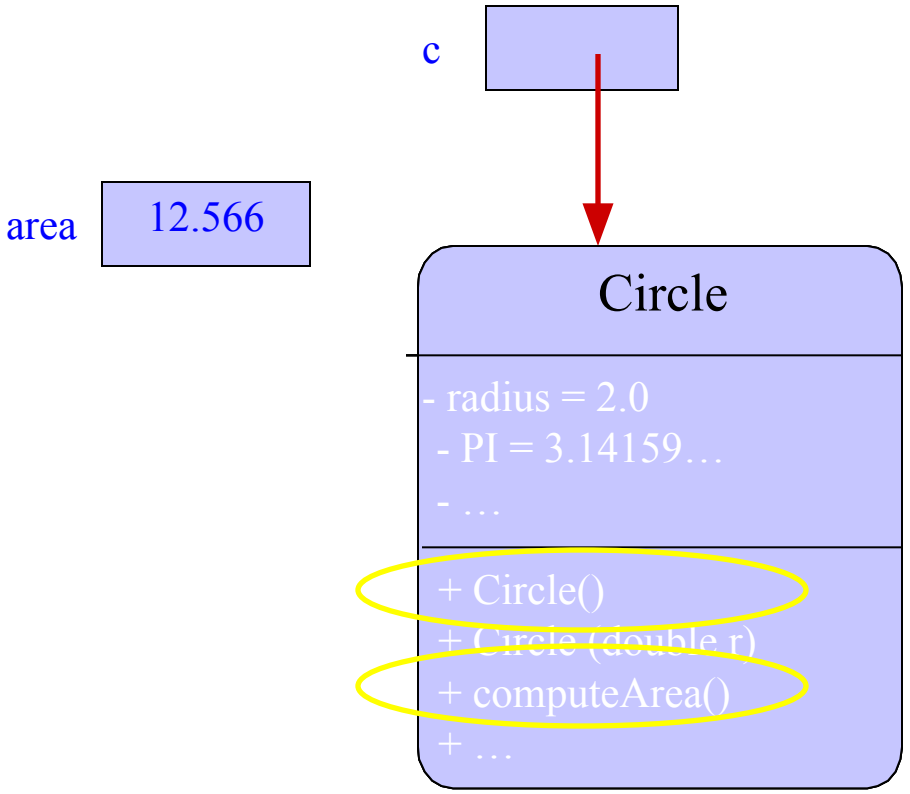
Using that method

```
public class CircleTest {
        public static void main (String[] args) {
                Circle c = new Circle();
                c.radius = 2.0;
                double area = c.computeArea();
                System.out.println (area);
  }
}
```

Prints 12.566370614356

What happens when that method is called

```
public class Circle {
        double radius;
        final static double PI = 3.1415926536;

        public Circle() {
                radius = 1.0;
        }

        // other constructor

        double computeArea () {
      return PI*radius*radius;
    }

}

public class CircleTest {
        public static void main (String[] args) {
                Circle c = new Circle();
                c.radius = 2.0;
                double area = c.computeArea();
                System.out.println (area);
  }
}
```



A note about methods/variable order

- Within a method, a variable must be declared before it is used
- In a class, methods and variables can be declared in any order
  - This is different than C++

Motivation for private fields

- Problem: We do not want people using our Circle class to be able to modify the fields on their own
- Solution: Don't allow other code to modify the radius field
  - Give it private visibility
- private means that only code within the class can modify the field

Visibilities in Java

- There are four visibilities:
  - private: Only code within the same class can access the field or method
    - Note: "access" means reading or writing the field, or invoking the method
  - public: Any code, anywhere, can access the field or method
  - protected: Used with inheritance
  - default: Almost the same as public

A few notes on visibilities

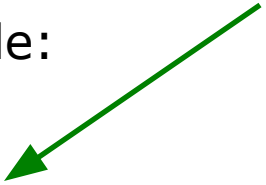
- You can NOT specify visibilities for method variables
  - Any method variable can only be accessed within that method

- You can also specify visibilities for methods and classes

Overriding methods (and constructors)

- Consider the following code:

```
Circle c1 = new Circle ();
Circle c2 = new Circle (2.0);
```



Creates a Circle of radius 1.0

Creates a Circle of radius 2.0

- Java knows which constructor to call by the list of parameters
  - This is called "overloading"
  - Meaning it means multiple things, depending on the context

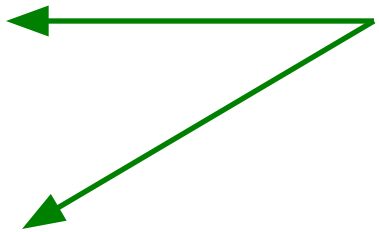
- We've seen overloading before:
  - 3+4 Performs integer addition
  - 3.0+4.0 Performs floating-point addition
  - "3"+"4" Performs string concatenation

- The '+' operator is overloaded

Overriding methods (and constructors), take 2

- The following Circle constructors would not be allowed:
  - We are assuming PI is not final for this example

```
public Circle() {
  radius = 1.0;
}

public Circle (double r) {
  radius = r;
}

public Circle (double p) {
  PI = p;
}
```



When Circle(1.0) is called, which one is meant?

Back to the static discussion

- Remember that there is one (and only one) static PI field, regardless of how many objects are declared
- Consider the following method:

```
double getPI() {
  return PI;
}
```

- It doesn't read or modify the "state" of any object
  - In this example, it doesn't read/write the radius
- In fact, that particular method doesn't care anything about the objects declared
  - It's only accessing a static field

Make getPI() static

- Consider the following:

```
static double getPI() {
    return PI;
}
```

- As the method is static, it can ONLY access static fields

- A static method does not care about the "state" of an object
  - Examples: Math.sin(), Math.tan(), Math.cos()
    - They don't care about the state of any Math object
    - They only perform the computation

Invoking static methods

- As with static fields, they can be called using either an object or the class name:

```
Circle c = new Circle();
System.out.println (c.getPI());
System.out.println (Circle.getPI());
```

- Static methods are also called class methods

static methods and non-static fields

- Consider the following (illegal) Circle method:

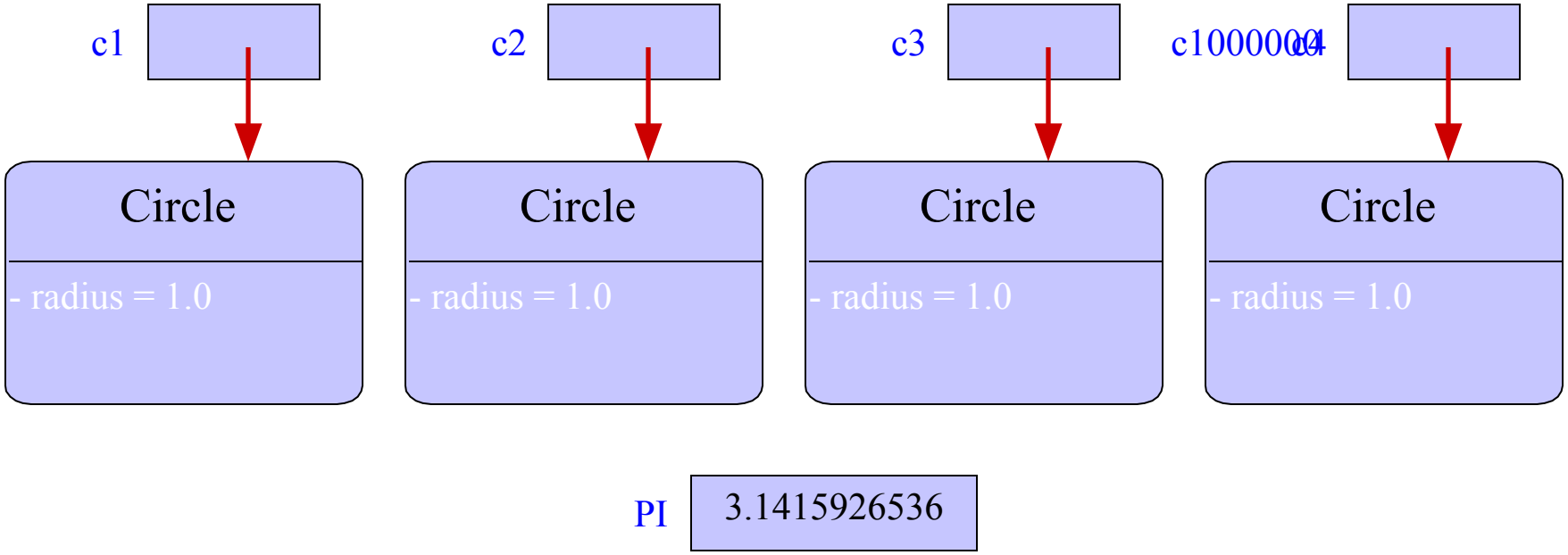
```
static double getRadius() {
        return radius;
      }
```

- And the code to invoke it:

```
public static void main (String[] args) {
  Circle c1 = new Circle();
  Circle c2 = new Circle();
  Circle c3 = new Circle();
  Circle c4 = new Circle();
  System.out.println (Circle.getRadius());
}
```

What happening in memory

- There are 4 Circle objects in memory
- Which radius field does Circle.getRadius() want?



The main static lesson

- A static method cannot access or modify the state of the object it is a part of

- If you remember nothing else about static methods, remember this!

static and non-static rules

- Non-static fields and methods can ONLY be accessed by the object name
- Static fields and methods can be accessed by EITHER the class name or the object name
- Non-static methods can refer to BOTH static and non-static fields
- Static methods can ONLY access static fields of the class they are part of

Back to our main() method

The method does not return a value

Any code anywhere
can call this method

It's a static method:

- Can't access non-static fields or methods directly
- Can be called only by the class name

Implications of main() being static

- It can call other static methods within the same class

```
class StaticMethods {
   static void method1() {
      System.out.println ("hi!");
   }

   public static void main (String args[]) {
      method1();
   }
}
```

- Note that we didn't have to prefix method1() with a object
  - Java assumes that it is in the same class

Inheritance and Polymorphism

Motivation

- Consider a transportation computer game
  - Different types of vehicles:
    - Planes
      - Jets, helicopters, space shuttle
    - Automobiles
      - Cars, trucks, motorcycles
    - Trains
      - Diesel, electric, monorail
    - Ships
      - …
- Let’s assume a class is written for each type of vehicle

Motivation

- Sample code for the types of planes:
  - fly()
  - takeOff()
  - land()
  - setAltitude()
  - setPitch()
- Note that a lot of this code is common to all types of planes
  - They have a lot in common!
  - It would be a waste to have to write separate fly() methods for each plane type
    - What if you then have to change one – you would then have to change dozens of methods

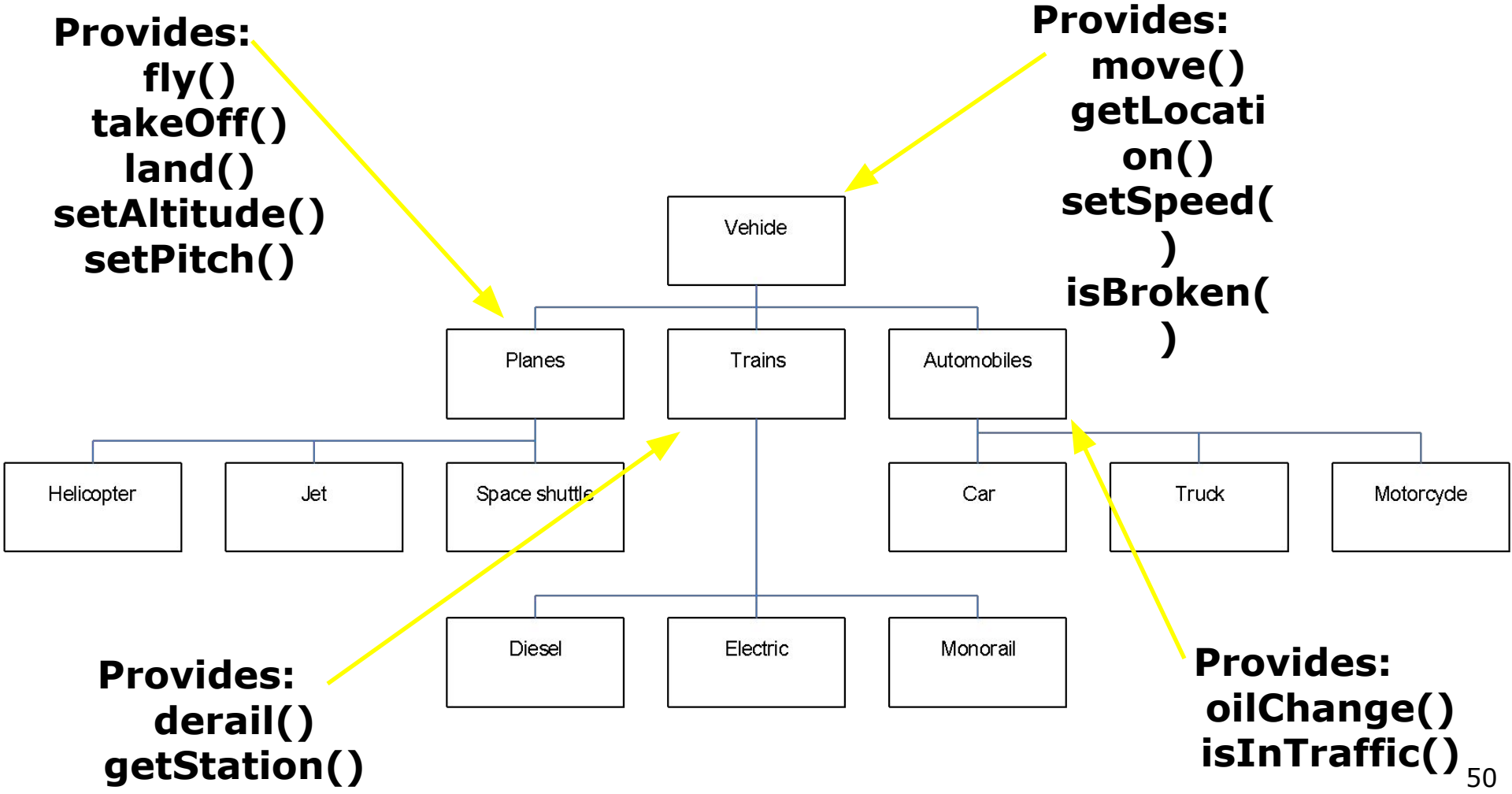
Motivation

- Indeed, all vehicles will have similar methods:
  - move()
  - getLocation()
  - setSpeed()
  - isBroken()

- Again, a lot of this code is common to all types of vehicles
  - It would be a waste to have to write separate move() methods for each vehicle type
    - What if you then have to change one – you would then have to change dozens of methods

- What we want is a means to specify one move() method, and have each vehicle type ***inherit*** that code
  - Then, if we have to change it, we only have to change one copy

Motivation


Provides:
fly()
takeOff()
land()
setAltitude()
setPitch()
Provides:
move()
getLocation()
setSpeed()
isBroken()
Vehide
Planes
Trains
Automobiles
Helicopter
Jet
Space shuttle
Car
Truck
Motorcycle
Diesel
Electric
Monorail
Provides:
derail()
getStation()
Provides:
oilChange()
isInTraffic()

Motivation

- What we will do is create a “parent” class and a “child” class
- The “child” class (or subclass) will inherit the methods (etc.) from the “parent” class (or superclass)
- Note that some classes (such as Train) are both subclasses and superclasses

Inheritance code

```
class Vehicle {
   ...
}

class Train extends Vehicles {
 ...
}

class Monorail extends Train {
 ...
}
```

About extends

- If class A extends class B
  - Then class A is the subclass of B
  - Class B is the superclass of class A
  - A "is a" B
  - A has (almost) all the methods and variables that B has

- If class Train extends class Vehicle
  - Then class Train is the subclass of Vehicle
  - Class Vehicle is the superclass of class Train
  - Train "is a" Vehicle
  - Train has (almost) all the methods and variables that Vehicle has

Object-oriented terminology

- In object-oriented programming languages, a class created by extending another class is called a *subclass*
- The class used for the basis is called the *superclass*
- Alternative terminology
  - The superclass is also referred to as the *base* class
  - The subclass is also referred to as the *derived* class