

Advanced Object-oriented Programming

Lecture 2

What is a Class?



Object-oriented Languages

Object-oriented languages are designed to facilitate structuring code at high levels of abstraction.

One of the key features of these languages is the ability to structure code at the level of *classes*.

In Java,

- all¹ code occurs in a **method**,
- all methods belong to a **class**
- all classes belong to a **package**².

1. Almost! 2. We won't go into packages in much detail

Object-oriented Languages

Object-oriented languages are designed to facilitate structuring code at high levels of abstraction.

One of the key features of these languages is the ability to structure code at the level of *classes*.

In Java,

- all¹ code occurs in a **method**,
- all methods belong to a **class**
- all classes belong to a **package**².

1. Almost! 2. We won't go into packages in much detail

Object-oriented Languages

Object-oriented languages are designed to facilitate structuring code at high levels of abstraction.

One of the key features of these languages is the ability to structure code at the level of *classes*.

In Java,

- **all**¹ code occurs in a **method**,
- all methods belong to a **class**
- all classes belong to a **package**².

1. Almost! 2. We won't go into packages in much detail

Object-oriented Languages

Object-oriented languages are designed to facilitate structuring code at high levels of abstraction.

One of the key features of these languages is the ability to structure code at the level of *classes*.

In Java,

- all¹ code occurs in a **method**,
- all methods belong to a **class**
- all classes belong to a **package**².

1. Almost! 2. We won't go into packages in much detail

Why Structure Code?

The main reason is that methods, classes and packages serve to group 'related' bits of code;

bringing related bits of code together makes it easier to:

- *maintain* that code;
- *(re)use* that code; and
- (once you've got the hang of it!) *write* that code.

This means 'uses related data' — finding the relationships between data is part of the craft of a programmer.

Why Structure Code?

The main reason is that methods, classes and packages serve to group 'related' bits of code;

bringing related bits of code together makes it easier to:

- *maintain* that code;
- *(re)use* that code; and
- (once you've got the hang of it!) *write* that code.

This means 'uses related data' — finding the relationships between data is part of the craft of a programmer.

Why Structure Code?

The main reason is that methods, classes and packages serve to group **'related'** bits of code;

bringing related bits of code together makes it easier to:

- *maintain* that code;
- *(re)use* that code; and
- (once you've got the hang of it!) *write* that code.

This means **'uses related data'** — finding the relationships between data is part of the craft of a programmer.

Why Structure Code?

The main reason is that methods, classes and packages serve to group 'related' bits of code;

bringing related bits of code together makes it easier to:

- *maintain* that code;
- *(re)use* that code; and
- (once you've got the hang of it!) *write* that code.

This means 'uses related data' — finding the relationships between data is part of the craft of a programmer.

Why Structure Code?

The main reason is that methods, classes and packages serve to group 'related' bits of code;

bringing related bits of code together makes it easier to:

- *maintain* that code;
- *(re)use* that code; and
- (once you've got the hang of it!) *write* that code.

This means 'uses related data' — finding the relationships between data is part of the craft of a programmer.

Why Structure Code?

The main reason is that methods, classes and packages serve to group 'related' bits of code;

bringing related bits of code together makes it easier to:

- *maintain* that code;
- *(re)use* that code; and
- (once you've got the hang of it!) *write* that code.

This means 'uses related data' — finding the relationships between data is part of the craft of a programmer.

Why Structure Code?

The main reason is that methods, classes and packages serve to group 'related' bits of code;

bringing related bits of code together makes it easier to:

- *maintain* that code;
- *(re)use* that code; and
- (once you've got the hang of it!) *write* that code.

This means 'uses related data' — finding the relationships between data is part of the craft of a programmer.

What is a Class?

In a very precise sense, classes are:

- collections of methods and fields
- types

less precisely, classes

- correspond to meaningful (kinds of) things.

(Many introductions to OO concentrate on this latter notion of class, which can be vague and confusing.)

What is a Class?

In a very precise sense, classes are:

- collections of methods and fields
- types

less precisely, classes

- correspond to meaningful (kinds of) things.

(Many introductions to OO concentrate on this latter notion of class, which can be vague and confusing.)

What is a Class?

In a very precise sense, classes are:

- collections of methods and fields
- types

less precisely, classes

- correspond to meaningful (kinds of) things.

(Many introductions to OO concentrate on this latter notion of class, which can be vague and confusing.)

What is a Class?

In a very precise sense, classes are:

- collections of methods and fields
- types

less precisely, classes

- correspond to meaningful (kinds of) things.

(Many introductions to OO concentrate on this latter notion of class, which can be vague and confusing.)

Classes as Types

In Java, all variables and expressions are *typed*, and the compiler checks for typing errors.

For example

```
int i = 5;  
String s = "ab";  
System.out.println(s * i);
```

will cause the compiler to report a type error:

Compiler output

```
TypeErr.java:7: operator * cannot be applied  
to java.lang.String,int  
    System.out.println(s * i);
```

(multiplication (*) requires two *ints* as arguments.)

int is the *type* of *i* *String* is the *type* of *s*

Classes as Types

In Java, all variables and expressions are *typed*, and the compiler checks for typing errors.

For example

```
int i = 5;  
String s = "ab";  
System.out.println(s * i);
```

will cause the compiler to report a type error:

Compiler output

```
TypeErr.java:7: operator * cannot be applied  
to java.lang.String,int  
    System.out.println(s * i);
```

(multiplication (*) requires two *ints* as arguments.)

int is the *type* of *i* *String* is the *type* of *s*

Classes as Types

In Java, all variables and expressions are *typed*, and the compiler checks for typing errors.

For example

```
int i = 5;  
String s = "ab";  
System.out.println(s * i);
```

will cause the compiler to report a type error:

Compiler output

```
TypeErr.java:7: operator * cannot be applied  
to java.lang.String,int  
    System.out.println(s * i);
```

(multiplication (*) requires two *ints* as arguments.)

int is the *type* of *i* *String* is the *type* of *s*

Classes as Types

In Java, all variables and expressions are *typed*, and the compiler checks for typing errors.

For example

```
int i = 5;  
String s = "ab";  
System.out.println(s * i);
```

will cause the compiler to report a type error:

Compiler output

```
TypeErr.java:7: operator * cannot be applied  
to java.lang.String,int  
    System.out.println(s * i);
```

(multiplication (*) requires two *ints* as arguments.)

int is the *type* of *i* *String* is the *type* of *s*

Classes as Types

In Java, all variables and expressions are *typed*, and the compiler checks for typing errors.

For example

```
int i = 5;  
String s = "ab";  
System.out.println(s * i);
```

will cause the compiler to report a type error:

Compiler output

```
TypeErr.java:7: operator * cannot be applied  
to java.lang.String,int  
    System.out.println(s * i);
```

(multiplication (*) requires two *ints* as arguments.)

int is the *type* of *i* **String** is the *type* of *s*

Classes as Types

In Java, all variables and expressions are *typed*, and the compiler checks for typing errors.

For example

```
int i = 5;  
String s = "ab";  
System.out.println(s * i);
```

will cause the compiler to report a type error:

Compiler output

```
TypeErr.java:7: operator * cannot be applied  
to java.lang.String,int  
    System.out.println(s * i);
```

(multiplication (*) requires two *ints* as arguments.)

int is the *type* of *i* *String* is the *type* of *s*

Classes as Types

In Java, all variables and expressions are *typed*, and the compiler checks for typing errors.

For example

```
int i = 5;  
String s = "ab";  
System.out.println(s * i);
```

will cause the compiler to report a type error:

Compiler output

```
TypeErr.java:7: operator * cannot be applied  
to java.lang.String,int  
    System.out.println(s * i);
```

(multiplication (*) requires two *ints* as arguments.)

int is the *type* of *i* *String* is the *type* of *s*

Classes as Types

In Java, all variables and expressions are *typed*, and the compiler checks for typing errors.

For example

```
int i = 5;  
String s = "ab";  
System.out.println(s * i);
```

will cause the compiler to report a type error:

Compiler output

```
TypeErr.java:7: operator * cannot be applied  
to java.lang.String,int
```

```
System.out.println(s * i);
```

(multiplication (*) requires two *ints* as arguments.)

int is the *type* of *i* *String* is the *type* of *s*

An Example Class

```
class Point
{
    int xCoord;
    int yCoord;

    Point(int x, int y)
    {
        xCoord = x;
        yCoord = y;
    }

    void move(int dx, int dy)
    {
        xCoord += dx;
        yCoord += dy;
    }
}
```

An Example Class

Class **Point** has:

- two **fields**:
 - `int` xCoord,
 - `int` yCoord
- one **constructor**:
 - `Point(int,int)`
- one **method**:
 - `void` move(int,int)

Type information in **red**; name in **blue**.

The fields and methods (*not* the constructors) are called the **members** of the class.

An Example Class

Class **Point** has:

- two **fields**:
 - **int** xCoord,
 - **int** yCoord
- one **constructor**:
 - Point(int,int)
- one **method**:
 - void move(int,int)

Type information in **red**; name in **blue**.

The fields and methods (*not* the constructors) are called the **members** of the class.

An Example Class

Class **Point** has:

- two **fields**:
 - **int** xCoord,
 - **int** yCoord
- one **constructor**:
 - **Point(int,int)**
- one **method**:
 - **void** move(int,int)

Type information in **red**; name in **blue**.

The fields and methods (*not* the constructors) are called the **members** of the class.

An Example Class

Class **Point** has:

- two **fields**:
 - **int** xCoord,
 - **int** yCoord
- one **constructor**:
 - **Point**(int,int)
- one **method**:
 - **void** move(int,int)

Type information in **red**; name in **blue**.

The fields and methods (*not* the constructors) are called the **members** of the class.

An Example Class

Class **Point** has:

- two **fields**:
 - `int` xCoord,
 - `int` yCoord
- one **constructor**:
 - `Point(int,int)`
- one **method**:
 - `void` move(int,int)

Type information in red; name in blue.

The fields and methods (*not* the constructors) are called the **members** of the class.

Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

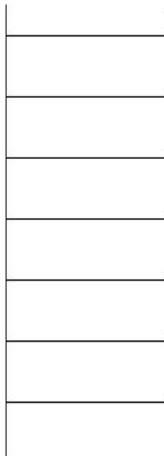
```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

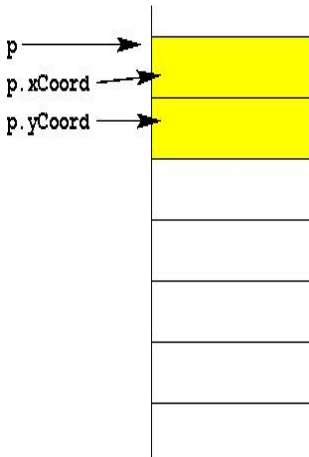


Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

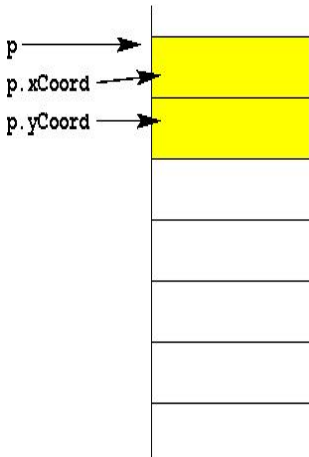


Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

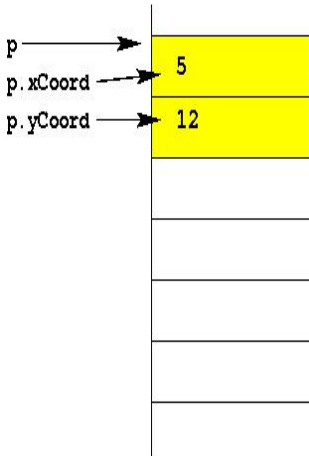


Programming with Points

```
Point p = new Point(5, 12);  
Point q = new Point(0, 0);  
q = p;  
q.move(2, 1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

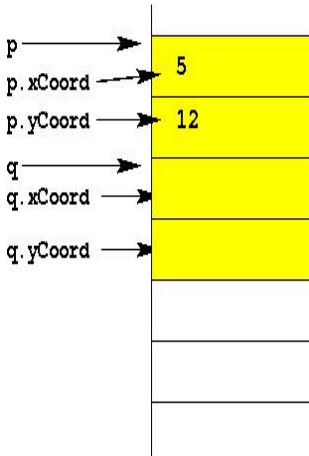


Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

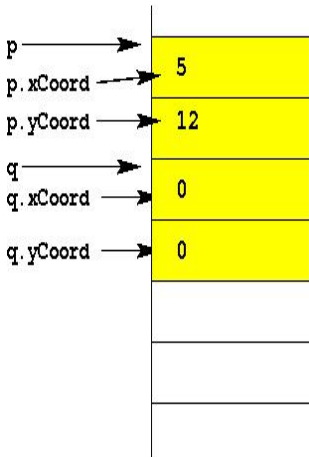


Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

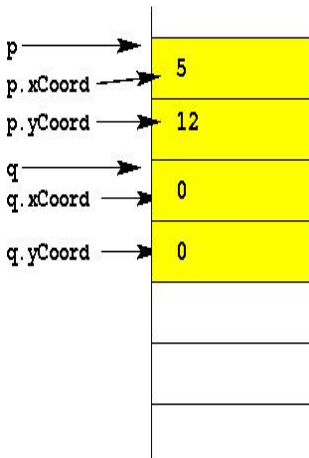


Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

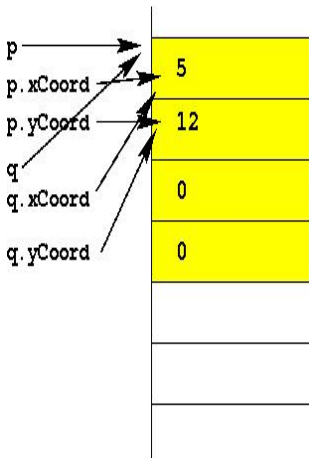


Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

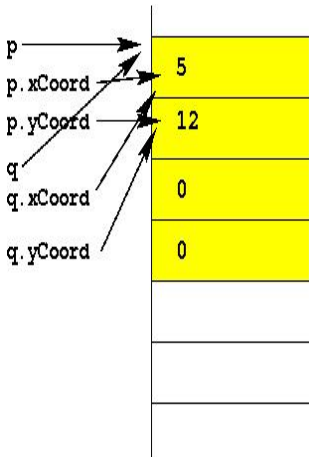


Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

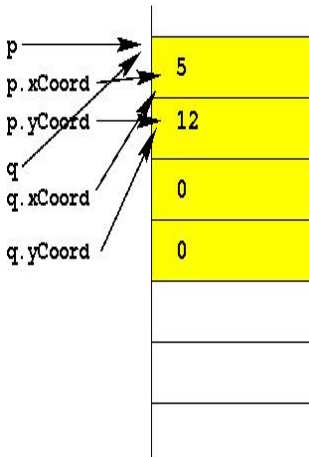


Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```

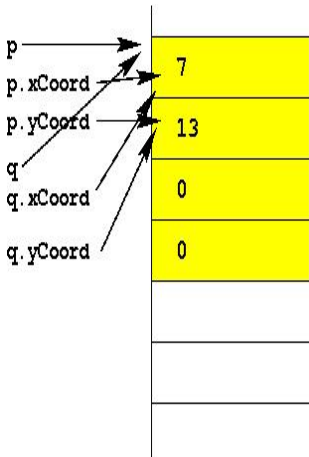


Programming with Points

```
Point p = new Point(5,12);  
Point q = new Point(0,0);  
q = p;  
q.move(2,1);
```

```
Point(int x, int y)  
{  
    xCoord = x;  
    yCoord = y;  
}
```

```
void move(int dx, int dy)  
{  
    xCoord += dx;  
    yCoord += dy;  
}
```



Classes as Types Again

```
Point p = new Point(5,12);  
p.move(1,2);  
p.println("oops"); // type error
```

The type error occurs at compile-time because class `Point` doesn't have a method `println(String)`.

Compiler output

```
Point.java:23: cannot find symbol  
symbol : method println(java.lang.String)  
location: class Point  
    p.println("oops");
```

Translation: huh?

Classes as Types Again

```
Point p = new Point(5,12);  
p.move(1,2);  
p.println("oops"); // type error
```

The type error occurs at compile-time because class `Point` doesn't have a method `println(String)`.

Compiler output

```
Point.java:23: cannot find symbol  
symbol : method println(java.lang.String)  
location: class Point  
    p.println("oops");
```

Translation: huh?

Classes as Types Again

```
Point p = new Point(5,12);  
p.move(1,2);  
p.println("oops"); // type error
```

The type error occurs at compile-time because class `Point` doesn't have a method `println(String)`.

Compiler output

```
Point.java:23: cannot find symbol  
symbol : method println(java.lang.String)  
location: class Point  
    p.println("oops");
```

Translation: huh?

Classes as Types Again

```
Point p = new Point(5,12);  
p.move(1,2);  
p.println("oops"); // type error
```

The type error occurs at compile-time because class `Point` doesn't have a method `println(String)`.

Compiler output

```
Point.java:23: cannot find symbol  
symbol : method println(java.lang.String)  
location: class Point  
    p.println("oops");
```

Translation: huh?

Classes as Types Again

```
Point p = new Point(5,12);  
p.move(1,2);  
p.println("oops"); // type error
```

The type error occurs at compile-time because class **Point** doesn't have a method **println(String)**.

Compiler output

```
Point.java:23: cannot find symbol  
symbol : method println(java.lang.String)  
location: class Point  
    p.println("oops");
```

Translation: huh?

Classes as Types Again

```
Point p = new Point(5,12);  
p.move(1,2);  
p.println("oops"); // type error
```

The type error occurs at compile-time because class `Point` doesn't have a method `println(String)`.

Compiler output

```
Point.java:23: cannot find symbol  
symbol : method println(java.lang.String)  
location: class Point  
    p.println("oops");
```

Translation: huh?

Inheritance

Inheritance allows methods and attributes declared in one class to be 'copied' into another.

This reduces the number of lines of code that a programmer needs to write, and is an example of **code reuse**.

The effect is much the same as copying and pasting the methods and fields of one class into another.

Inheritance

Inheritance allows methods and attributes declared in one class to be 'copied' into another.

This reduces the number of lines of code that a programmer needs to write, and is an example of **code reuse**.

The effect is much the same as copying and pasting the methods and fields of one class into another.

Inheritance

Inheritance allows methods and attributes declared in one class to be 'copied' into another.

This reduces the number of lines of code that a programmer needs to write, and is an example of **code reuse**.

The effect is much the same as copying and pasting the methods and fields of one class into another.

Inheritance

Inheritance allows methods and attributes declared in one class to be 'copied' into another.

This reduces the number of lines of code that a programmer needs to write, and is an example of **code reuse**.

The effect is much the same as copying and pasting the methods and fields of one class into another.

An Example of Inheritance

class LabelledPoint

```
class LabelledPoint extends Point {  
    String label;  
  
    LabelledPoint(int x, int y, String s) {  
        xCoord = x;  
        yCoord = y;  
        label = s;  
    }  
  
    void setLabel(String s) {  
        label = s;  
    }  
}
```

An Example of Inheritance

class LabelledPoint

```
class LabelledPoint extends Point {  
    String label;  
  
    LabelledPoint(int x, int y, String s) {  
        xCoord = x;  
        yCoord = y;  
        label = s;  
    }  
  
    void setLabel(String s) {  
        label = s;  
    }  
}
```

An Example of Inheritance

Class `LabelledPoint` has:

- *three* fields:
 - `int` `xCoord`, (inherited)
 - `int` `yCoord`, (inherited)
 - `String` `label` ('local')
- *one* constructor
 - `LabelledPoint(int,int,String)`
- *two* methods
 - `void` `move(int,int)` (inherited)
 - `void` `setLabel(String)` (local)

Note that the `Point` constructor is *not* inherited.

An Example of Inheritance

Class `LabelledPoint` has:

- *three* fields:
 - `int` `xCoord`, (inherited)
 - `int` `yCoord`, (inherited)
 - `String` `label` ('local')
- *one* constructor
 - `LabelledPoint(int,int,String)`
- *two* methods
 - `void` `move(int,int)` (inherited)
 - `void` `setLabel(String)` (local)

Note that the `Point` constructor is *not* inherited.

An Example of Inheritance

Class `LabelledPoint` has:

- *three* fields:
 - `int` `xCoord`, (inherited)
 - `int` `yCoord`, (inherited)
 - `String` `label` ('local')
- *one* constructor
 - `LabelledPoint(int,int,String)`
- *two* methods
 - `void` `move(int,int)` (inherited)
 - `void` `setLabel(String)` (local)

Note that the `Point` constructor is *not* inherited.

An Example of Inheritance

Class `LabelledPoint` has:

- *three* fields:
 - `int` `xCoord`, (inherited)
 - `int` `yCoord`, (inherited)
 - `String` `label` ('local')
- *one* constructor
 - `LabelledPoint(int,int,String)`
- *two* methods
 - `void` `move(int,int)` (inherited)
 - `void` `setLabel(String)` (local)

Note that the `Point` constructor is *not* inherited.

An Example of Inheritance

Class `LabelledPoint` has:

- *three* fields:
 - `int` `xCoord`, (inherited)
 - `int` `yCoord`, (inherited)
 - `String` `label` ('local')
- *one* constructor
 - `LabelledPoint(int,int,String)`
- *two* methods
 - `void` `move(int,int)` (inherited)
 - `void` `setLabel(String)` (local)

Note that the `Point` constructor is *not* inherited.

Classes as Types, Yet Again

Instances of subclasses can be used wherever an instance of the superclass can be used, **not vice-versa**.

```
Point p = new LabelledPoint(5,12,"blue");  
p.move(4,0);  
p.setLabel("red"); // type error  
LabelledPoint lp = new Point(2,2); // error
```

Compiler output

```
Point.java:47: cannot find symbol  
symbol : method setLabel(java.lang.String)  
location: class Point [...]  
Point.java:48: Incompatible types  
found   : Point  
required: LabelledPoint [...]
```

Classes as Types, Yet Again

Instances of subclasses can be used wherever an instance of the superclass can be used, **not vice-versa**.

```
Point p = new LabelledPoint(5,12,"blue");  
p.move(4,0);  
p.setLabel("red"); // type error  
LabelledPoint lp = new Point(2,2); // error
```

Compiler output

```
Point.java:47: cannot find symbol  
symbol : method setLabel(java.lang.String)  
location: class Point [...]  
Point.java:48: incompatible types  
found : Point  
required: LabelledPoint [...]
```

Classes as Types, Yet Again

Instances of subclasses can be used wherever an instance of the superclass can be used, **not vice-versa**.

```
Point p = new LabelledPoint(5,12,"blue");  
p.move(4,0);  
p.setLabel("red"); // type error  
LabelledPoint lp = new Point(2,2); // error
```

Compiler output

```
Point.java:47: cannot find symbol  
symbol : method setLabel(java.lang.String)  
location: class Point [...]  
Point.java:48: incompatible types  
found : Point  
required: LabelledPoint [...]
```

Classes as Types, Yet Again

Instances of subclasses can be used wherever an instance of the superclass can be used, **not vice-versa**.

```
Point p = new LabelledPoint(5,12,"blue");  
p.move(4,0);  
p.setLabel("red"); // type error  
LabelledPoint lp = new Point(2,2); // error
```

Compiler output

```
Point.java:47: cannot find symbol  
symbol : method setLabel(java.lang.String)  
location: class Point [...]  
Point.java:48: incompatible types  
found : Point  
required: LabelledPoint [...]
```

Classes as Types, Yet Again

Instances of subclasses can be used wherever an instance of the superclass can be used, **not vice-versa**.

```
Point p = new LabelledPoint(5,12,"blue");  
p.move(4,0);  
p.setLabel("red"); // type error  
LabelledPoint lp = new Point(2,2); // error
```

Compiler output

```
Point.java:47: cannot find symbol  
symbol : method setLabel(java.lang.String)  
location: class Point [...]  
Point.java:48: incompatible types  
found : Point  
required: LabelledPoint [...]
```

Classes as Types, Yet Again

Instances of subclasses can be used wherever an instance of the superclass can be used, **not vice-versa**.

```
Point p = new LabelledPoint(5,12,"blue");  
p.move(4,0);  
p.setLabel("red"); // type error  
LabelledPoint lp = new Point(2,2); // error
```

Compiler output

```
Point.java:47: cannot find symbol  
symbol : method setLabel(java.lang.String)  
location: class Point [...]  
Point.java:48: incompatible types  
found : Point  
required: LabelledPoint [...]
```

Classes as Thingummies

A **subclass** adds **members** (methods, attributes) to its **superclass**:

a **LabelledPoint** **is a** **Point** with a **label** and a method **setLabel()**
it isn't a *special case* of a 'point'

Compare:

- a square **is a** rectangle whose width and height are the same.

Exercise: try this in Java!

It won't work.

Classes as Thingummies

A **subclass** adds **members** (methods, attributes) to its **superclass**:

a **LabelledPoint** **is a** **Point** with a **label** and a method **setLabel()**
it isn't a *special case* of a 'point'

Compare:

- a square **is a** rectangle whose width and height are the same.

Exercise: try this in Java!

It won't work.

Things and Thingummies

A class should correspond to either some meaningful kind of entity in the application domain, or to an abstract data type (i.e., some way of ordering and grouping data, together with some methods for working with that data).

In the forthcoming lectures, we'll look in more detail at ADTs, classes, and things — not thingummies.

That's All, Folks!

Summary

- Classes are types
- Classes are their members

Next:

Fields and
Abstract Data Types

