Advanced Object-oriented Programming

Lecture 6

Scope
A class contains **declarations** of **members**. Members can be:

- fields (including constants)
- methods
- classes

Each of these has a **name**, as do classes themselves. Every name has a **scope**, which determines where the name may be used.
Scope of Variables

A variable may be local to a block of code:

```java
if (x > 10)
{
    int i = x;
    for (i; i > 10; i--)
    {
        System.out.println(report[i]);
    }
}
else
{
    i = 0; // compile-time error
}
```
A variable may be local to a block of code:

```java
if (x > 10)
{
    int i = x;
    for (i; i > 10; i--)
    {
        System.out.println(report[i]);
    }
}
else
{
    i = 0; // compile-time error
}
```
A variable may be local to a block of code:

```java
if (x > 10) {
    int i = x;
    for (i; i > 10; i--) {
        System.out.println(report[i]);
    }
} else {
    i = 0; // compile-time error
}
```
Scope of Variables

A variable may be local to a block of code:

```java
if (x > 10) {
    int i = x;
    for (i; i > 10; i--)
    {
        System.out.println(report[i]);
    }
}
else {
    i = 0; // compile-time error
}
```
Scope of Variables

A variable may be local to a block of code:

```java
if (x > 10)
{
    int i = x;
    for (i; i > 10; i--)
    {
        System.out.println(report[i]);
    }
}
else
{
    i = 0; // compile-time error
}
```
A variable may be local to a block of code:

```java
if (x > 10)
{
    int i = x;
    for (i; i > 10; i--)
    {
        System.out.println(report[i]);
    }
}
else
{
    i = 0;  // compile-time error
}
```
A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
    {
        System.out.println(report[i++]);
    }
}
```

```java
public void sillyMethod() {
    i = 0;  // compile-time error
}
```

Note also that `n` is local to the method
Scope of Variables

A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
        System.out.println(report[i++]);
}

public void sillyMethod() {
    i = 0;  // compile-time error
}
```

Note also that `n` is local to the method.
Scope of Variables

A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n) {
        System.out.println(report[i++]);
    }
}

public void sillyMethod() {
    i = 0; // compile-time error
}
```

Note also that `n` is local to the method
A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
    { System.out.println(report[i++]); } 
}

public void sillyMethod() {
    i = 0; // compile-time error
}
```

Note also that n is local to the method.
A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
    {
        System.out.println(report[i++]);
    }
}

public void sillyMethod() {
    i = 0; // compile-time error
}
```

Note also that `n` is local to the method.
A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
    {
        System.out.println(report[i++]);
    }
}

public void sillyMethod() {
    i = 0;  // compile-time error
}
```

Note also that $n$ is local to the method.
Scope of Variables

A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
        { System.out.println(report[i++]); } 
}

public void sillyMethod() {
    i = n;    // compile-time error  
}
```

Note also that `n` is local to the method
A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
    {
        System.out.println(report[i++]);
    }
}

public void sillyMethod() {
    i = n;    // compile-time error
}
```

Note also that \( n \) is local to the method
A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
        { System.out.println(report[i++]); } 
}

public void sillyMethod() {
    i = n; // compile-time error
}
```

Note also that `n` is local to the method.
Scope of Variables

A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
    {
        System.out.println(report[i++]);
    }
}

public void sillyMethod() {
    i = n;  // compile-time error
}
```

Note also that n is local to the method
Scope of Variables

A variable may be local to a method:

```java
public void printTo(int n) {
    int i = 0;
    while (i < n)
    { System.out.println(report[i++]); }
}

public void sillyMethod() {
    i = n; // compile-time error
}
```

Note also that n is local to the method
Debugging Tip!

A common (and frustrating, because it’s hard to spot) error:

```java
class Point {
    int xCoord; // not assigned to

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

If you know something is wrong with a variable, then — in Emacs — Ctrl+s (search) or Ctrl+r (search backwards) can be useful (as well as patience and persistence).
Debugging Tip!

A common (and frustrating, because it’s hard to spot) error:

```java
class Point
{
    int xCoord; // not assigned to

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

If you know something is wrong with a variable, then — in Emacs — Ctrl+s (search) or Ctrl+r (search backwards) can be useful (as well as patience and persistence).
Debugging Tip!

A common (and frustrating, because it’s hard to spot) error:

class Point
{
    int xCoord;  // not assigned to
    Point(int x, int y)
    {
        int xCoord = x;
        ...
    }
    ...
}

If you know something is wrong with a variable, then — in Emacs — Ctrl+s (search) or Ctrl+r (search backwards) can be useful (as well as patience and persistence).
Debugging Tip!

A common (and frustrating, because it’s hard to spot) error:

```java
class Point {
    int xCoord; // not assigned to

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

If you know *something* is wrong with a variable, then — in Emacs — Ctrl+s (search) or Ctrl+r (search backwards) can be useful (as well as patience and persistence).
A common (and frustrating, because it’s hard to spot) error:

```java
class Point {
    int xCoord;  // not assigned to

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

If you know *something* is wrong with a variable, then — in Emacs — Ctrl+s (search) or Ctrl+r (search backwards) can be useful (as well as patience and persistence).
A variable may be declared in a class:

class Point
{
    int xCoord;
    ...
}

— in which case, it is a **field**.
The Scope of Fields

As declared in class `Point`, the field `xCoord` is ‘visible’ (in scope) throughout the class. In fact, it is visible throughout the package that class `Point` is in.

```java
class SomeOtherClass
{
    ...
    Point p = new Point(2,5);
    p.xCoord = 10;
    ...
}
```

The scope of fields, and members in general, can be controlled using the scope modifiers: `public`, `protected`, `private`, and ‘default’ (no scope modifier used).
The Scope of Fields

As declared in class `Point`, the field `xCoord` is ‘visible’ (in scope) throughout the class. In fact, it is visible throughout the package that class `Point` is in.

```java
class SomeOtherClass {
    ...
    Point p = new Point(2,5);
    p.xCoord = 10;
    ...
}
```

The scope of fields, and members in general, can be controlled using the **scope modifiers**: `public`, `protected`, `private`, and ‘default’ (no scope modifier used).
Why Restrict Scope?

The goal of programming is to implement a given *functionality*. This usually involves choosing some representation for the data used by the program, but the functionality is the important thing.

If the details of data representations (implementations) are hidden, then code can be more freely modified — but of course the code has to be modified in such a way that *the functionality is preserved*. 
Why Restrict Scope?

The goal of programming is to implement a given *functionality*. This usually involves choosing some representation for the data used by the program, but the functionality is the important thing.

If the details of data representations (implementations) are hidden, then code can be more freely modified — but of course the code has to be modified in such a way that the functionality is preserved.
Why Restrict Scope?

The goal of programming is to implement a given \textit{functionality}. This usually involves choosing some representation for the data used by the program, but the functionality is the important thing.

If the details of data representations (implementations) are hidden, then code can be more freely modified — but of course the code has to be modified in such a way that \textit{the functionality is preserved}.
Digression: mistakes are good

Changes to code are an unfortunate but understandable result of the fact that humans — including programmers — don’t have perfect insight.

A difficult problem may require a complex solution, and finding an elegant solution may require several false starts.

These false starts often provide the insight that leads to a better solution.

...then after working on that for a while, you start to see how it should have been done in the first place.

The term refactoring is perhaps an attempt to impose some respectability on this fitful lurching towards elegance.
Changes to code are an unfortunate but understandable result of the fact that humans — including programmers — don’t have perfect insight.

A difficult problem may require a complex solution, and finding an elegant solution may require several false starts.

These false starts often provide the insight that leads to a better solution.

... then after working on that for a while, you start to see how it should have been done in the first place.

The term refactoring is perhaps an attempt to impose some respectability on this fitful lurching towards elegance.
Digression: mistakes are good

Changes to code are an unfortunate but understandable result of the fact that humans — including programmers — don’t have perfect insight.

A difficult problem may require a complex solution, and finding an elegant solution may require several false starts.

These false starts often provide the insight that leads to a better solution.

... then after working on that for a while, you start to see how it should have been done in the first place.

The term refactoring is perhaps an attempt to impose some respectability on this fitful lurching towards elegance.
Digression: mistakes are good

Changes to code are an unfortunate but understandable result of the fact that humans — including programmers — don’t have perfect insight.

A difficult problem may require a complex solution, and finding an elegant solution may require several false starts.

These false starts often provide the insight that leads to a better solution.

...then after working on that for a while, you start to see how it should have been done in the first place.

The term **refactoring** is perhaps an attempt to impose some respectability on this fitful lurching towards elegance.
Why Restrict Scope?

Software changes (is ‘refactored’).

Very often one piece of software uses another piece of software, which uses another (and so on).

How many Java classes have you used that you haven’t written yourself?

String, Array, . . .

hopefully a lot more by the end of this module!

So one piece of software can depend upon many others.
Why Restrict Scope?

Software changes (is ‘refactored’).

Very often one piece of software uses another piece of software, which uses another (and so on).

How many Java classes have you used that you haven’t written yourself?

**String, Array, ...**

hopefully a lot more by the end of this module!

So one piece of software can depend upon many others.
Why Restrict Scope?

Software changes (is ‘refactored’).

Very often one piece of software uses another piece of software, which uses another (and so on).

How many Java classes have you used that you haven’t written yourself?

String, Array, . . .

hopefully a lot more by the end of this module!

So one piece of software can depend upon many others.
Why Restrict Scope?

Programs use data: input, output, and ‘intermediate values’ are all data, and all data is represented in a particular way e.g., a sequence of values might be represented by an array. But programs also work with this data — this is the functionality.

It turns out that data representations change more often than functionality.

We can protect software that uses our software from any changes we may make by hiding the details of data representation.

How? — by using scope modifiers (especially private).
Why Restrict Scope?

Programs use data: input, output, and ‘intermediate values’ are all data, and all data is represented in a particular way e.g., a sequence of values might be represented by an array.

But programs also work with this data — this is the functionality.

It turns out that data representations change more often than functionality.

We can protect software that uses our software from any changes we may make by hiding the details of data representation.

How? — by using scope modifiers (especially private).
Why Restrict Scope?

Programs use data: input, output, and ‘intermediate values’ are all data, and all data is represented in a particular way e.g., a sequence of values might be represented by an array. But programs also work with this data — this is the functionality.

It turns out that data representations change more often than functionality.

We can protect software that uses our software from any changes we may make by hiding the details of data representation.

How? — by using scope modifiers (especially private).
Why Restrict Scope?

Programs use data: input, output, and ‘intermediate values’ are all data, and all data is represented in a particular way e.g., a sequence of values might be represented by an array. But programs also work with this data — this is the functionality.

It turns out that data representations change more often than functionality.

We can protect software that uses our software from any changes we may make by hiding the details of data representation.

How? — by using scope modifiers (especially private).
Why Restrict Scope?

Programs use data: input, output, and ‘intermediate values’ are all data, and all data is represented in a particular way e.g., a sequence of values might be represented by an array. But programs also work with this data — this is the functionality.

It turns out that data representations change more often than functionality.

We can protect software that uses our software from any changes we may make by hiding the details of data representation.

How? — by using scope modifiers (especially private).
Data Representations and Functionality

Programming:

- we want to implement a given functionality;
- the data we use has to be represented somehow;
- but we want to hide the details of the representation we choose.

This works best if we have some precise way of describing both the functionality and the data representation.

For this, we shall use **Maude**: a notation for describing abstract data types.
Data Representations and Functionality

Programming:

- we want to implement a given functionality;
- the data we use has to be represented somehow;
- but we want to hide the details of the representation we choose.

This works best if we have some precise way of describing both the functionality and the data representation.

For this, we shall use **Maude**: a notation for describing abstract data types.
Data Representations and Functionality

Programming:

- we want to implement a given functionality;
- the data we use has to be represented somehow;
- but we want to hide the details of the representation we choose.

This works best if we have some precise way of describing both the functionality and the data representation.

For this, we shall use **Maude**: a notation for describing abstract data types.
Data Representations and Functionality

Programming:

- we want to implement a given functionality;
- the data we use has to be represented somehow;
- but we want to hide the details of the representation we choose.

This works best if we have some precise way of describing both the functionality and the data representation.

For this, we shall use Maude: a notation for describing abstract data types.
Data Representations and Functionality

Programming:
- we want to implement a given functionality;
- the data we use has to be represented somehow;
- but we want to hide the details of the representation we choose.

This works best if we have some precise way of describing both the functionality and the data representation.

For this, we shall use Maude: a notation for describing abstract data types.
Abstract Data Types

An abstract data type (ADT) consists of:

- a set of abstract data values, and
- some specified operations on those values.

— ‘Abstract’ means no particular data representation (implementation) is given.
Example ADT: Stacks

A **Stack** is a sequence of values with operations:

- **push** — add a value onto the top of the stack
- **top** — get the value at the top of the stack
- **pop** — remove the last value added to the stack
- **empty** — a constant representing a sequence with no values
A **Stack** is a sequence of values with operations:

- **push** — add a value onto the top of the stack
- **top** — get the value at the top of the stack
- **pop** — remove the last value added to the stack
- **empty** — a constant representing a sequence with no values
A Specification of Stacks

Maude

\[\begin{align*}
\text{fmod STACK is} \\
\quad \text{protecting} \ \text{INT} . \\
\quad \text{sort} \ \text{Stack} . \\
\quad \text{op} \ \text{push} : \text{Int} \ \text{Stack} \rightarrow \text{Stack} . \\
\quad \text{op} \ \text{top} : \text{Stack} \rightarrow \text{Int} . \\
\quad \text{op} \ \text{pop} : \text{Stack} \rightarrow \text{Stack} . \\
\quad \text{op} \ \text{empty} : \rightarrow \text{Stack} . \\
\quad \text{var} \ I : \text{Int} . \\
\quad \text{var} \ S : \text{Stack} . \\
\quad \text{eq} \ \text{top}(\text{push}(I, S)) = I . \\
\quad \text{eq} \ \text{pop}(\text{push}(I, S)) = S . \\
\text{endfm}
\end{align*}\]
A Specification of Stacks

Maude

fmod STACK is
    protecting INT.
    sort Stack.
    op push : Int Stack -> Stack.
    op top : Stack -> Int.
    op pop : Stack -> Stack.
    op empty : -> Stack.

    var I : Int.
    var S : Stack.
    eq top(push(I, S)) = I.
    eq pop(push(I, S)) = S.
endfm
A Specification of Stacks

fmod STACK is
  protecting INT .
  sort Stack .
  op push : Int Stack -> Stack .
  op top : Stack -> Int .
  op pop : Stack -> Stack .
  op empty : -> Stack .

  var I : Int .
  var S : Stack .
  eq top(push(I, S)) = I .
  eq pop(push(I, S)) = S .
endfm
States of Stacks

empty

Stack st = new Stack();
States of Stacks

push(3, empty)

Stack st = new Stack();
st.push(3);
States of Stacks

push(7, push(3, empty))

Stack st = new Stack();
st.push(3);
st.push(7);
States of Stacks

pop(push(7, push(3, empty)))

Stack st = new Stack();
st.push(3);
st.push(7);
st.pop();
States of Stacks

\text{pop(push(7, push(3, empty)))}

\textbf{var} \ l : \text{Int} . \\
\textbf{var} \ S : \text{Stack} . \\
\textbf{eq} \ \text{pop(push(l, S))} = S .

\text{Stack \ st} = \text{new Stack}(); \\
st.\text{push}(3); \\
st.\text{push}(7); \\
st.\text{pop}();
States of Stacks

\[
pop(push(7, push(3, empty))) = push(3, empty)
\]

\[
\text{var } l : \text{Int} .
\text{var } S : \text{Stack} .
\text{eq pop(push(l, S)) = S .}
\]

\[
\text{Stack } st = \text{new Stack()};
st.push(3);
st.push(7);
st.pop();
\]
States of Stacks

\[
pop(push(7, push(3, empty)))
\]
\[
push(3, empty)
\]

Stack \( st = new Stack() \);
\( st.push(3); \)
\( st.push(7); \)
\( st.pop(); \)
States of Stacks

\[
\text{push}(0, \text{pop}(\text{push}(7, \text{push}(3, \text{empty})))) \\
= \text{push}(0, \text{push}(3, \text{empty}))
\]

Stack \( st = \text{new Stack}() \);
\( st.\text{push}(3) \);
\( st.\text{push}(7) \);
\( st.\text{pop}() \);
\( st.\text{push}(0) \);
States of Stacks

\[
push(5, push(0, pop(push(7, push(3, empty)))))
= push(5, push(0, push(3, empty)))
\]

```java
Stack st = new Stack();
st.push(3);
st.push(7);
st.pop();
st.push(0);
st.push(5);
```
States of Stacks

pop( push(5, push(0, pop(push(7, push(3, empty)))))))
= pop(push(5, push(0, push(3, empty))))

Stack st = new Stack();
st.push(3);
st.push(7);
st.pop();
st.push(0);
st.push(5);
st.pop();
States of Stacks

pop( push(5, push(0, pop(push(7, push(3, empty)))))))
= pop(push(5, push(0, push(3, empty))))

var l : Int.
var S : Stack.
eq pop(push(l, S)) = S.

Stack st = new Stack();
st.push(3);
st.push(7);
st.pop();
st.push(0);
st.push(5);
st.pop();
States of Stacks

pop( push(5, push(0, pop(push(7, push(3, empty))))) )
= pop(push(5, push(0, push(3, empty))))
= push(0, push(3, empty))

```
var l : Int.
var S : Stack.
eq pop(push(l, S)) = S.
```

```
Stack st = new Stack();
st.push(3);
st.push(7);
st.pop();
st.push(0);
st.push(5);
st.pop();
```
An Implementation of Stacks

```java
public class Stack {
    private int[] values;
    private int pointer;

    public Stack() {
        values = new int[100];
        pointer = -1;
    }
}
```
An Implementation of Stacks

```java
public class Stack {

    private int[] values;
    private int pointer;

    public Stack() {
        values = new int[100];
        pointer = -1;
    }
}
```
An Implementation of Stacks

```java
public class Stack {

    private int[] values;
    private int pointer;

    public Stack() {
        values = new int[100];
        pointer = -1;
    }
}
```
public int top() {
    return values[pointer];
}

public void pop() {
    pointer--;
}

public void push(int v) {
    values[++pointer] = v;
}
public int top() {
    return values[pointer];
}

public void pop() {
    pointer--;
}

public void push(int v) {
    values[++pointer] = v;
}
class Stack contd.

// top : Stack -> Int
public int top() {
    return values[pointer];
}

// pop : Stack -> Stack
public void pop() {
    pointer--;
}

// push : Int Stack -> Stack
public void push(int v) {
    values[++pointer] = v;
}
}
Private is Hidden

Attempting to access a member outside its scope causes a compile-time error.

```java
public class SomeOtherClass {
    ...
    Stack st = ...;
    int noWay = st.values[0];
    ...
}
```
Attempting to access a member outside its scope causes a compile-time error.

```java
public class SomeOtherClass {
    ...
    Stack st = ...;
    int noWay = st.values[0];  // compiler error
    ...
}
```
Now that the representation details are hidden,

- classes that *use* class `Stack` must follow the Last-In, First-Out restriction on accessing elements of the stack, because they can only use the public methods `top`, `pop`, and `push`;
Now that the representation details are hidden,

- classes that *use* class Stack must follow the Last-In, First-Out restriction on accessing elements of the stack, because they can only use the public methods top, pop, and push;
- the ‘owner’ (programmer) of class Stack is free to change the data representation (e.g., using a linked list instead of an array to store the elements) — so long as the functionality (as specified in Maude) remains the same, client classes that use class Stack won’t notice any difference.
Scope of Classes

**public** — visible inside and outside its package (i.e., it can be imported into another package)

**default** — visible only inside its own package (i.e., it *cannot* be imported into another package)

**private** — visible only inside its own compilation unit (file: so it’s not visible in other classes in any other file).
Scope of Members

**public** — visible everywhere the class is visible

**default** — visible only within the package, wherever the class is visible

**private** — visible only within its class

**protected** — inside the package, visible wherever the class is; outside the package, visible only in subclasses
Hiding Data Representation

```
public class Point {
    private int xCoord;
    private int yCoord;
    public Point(int x, int y) {
        xCoord = x;
        yCoord = y;
    }
    public void move(int dx, int dy) { ... }
}
```
Hiding Data Representation

```java
public class Point {
    private int xCoord;
    private int yCoord;
    public Point(int x, int y) {
        xCoord = x;
        yCoord = y;
    }
    public void move(int dx, int dy) {
    }
}
```
Hiding Data Representation

```java
public class Point {
    private int xCoord;
    private int yCoord;
    public Point(int x, int y) {
        xCoord = x;
        yCoord = y;
    }
    public void move(int dx, int dy) {
    }
}
```
Hiding Data Representations

As a rule of thumb:

Principle 4

Members of a class (fields, methods, etc.) should be declared private
As a rule of thumb:

Principle 4

Members of a class (fields, methods, etc.) should be declared **private**

...unless there is a good reason not to!
public class LabelledPoint extends Point {
    private String label;
    public LabelledPoint(int x, int y, String s) {
        xCoord = x;
    }
}
But Now...

```java
public class LabelledPoint
    extends Point
{
    private String label;

    public LabelledPoint(int x, int y, String s)
    {
        xCoord = x;    // compile-time error
    }
```
Working with Hidden Information

We could change class Point:

```java
protected int xCoord;
```

—but why should `xCoord` be visible throughout the package?—this spoils our neat hiding of data representation.

In this case, there is another solution...
Working with Hidden Information

We could change class Point:

```java
    protected int xCoord;
```

—but why should `xCoord` be visible throughout the package?
We could change class Point:

```java
protected int xCoord;
```

— but why should `xCoord` be visible throughout the package?
— this spoils our neat hiding of data representation.
Working with Hidden Information

We could change class Point:

```java
protected int xCoord;
```

— but why should `xCood` be visible throughout the package?
— this spoils our neat hiding of data representation.

In this case, there is another solution. . . .
Recall that constructors aren’t inherited.
Recall that constructors aren’t inherited.
They can, however, be invoked using the `super` keyword.
Recall that constructors aren’t inherited. They can, however, be invoked using the `super` keyword. This refers to the constructor of the immediate superclass (in this case, class `Point`.)
public class LabelledPoint extends Point {
    private String label;
    public LabelledPoint(int x, int y, String s) {
        super(x,y);
        label = s;
    }
    ...
}
Constructors are typically used to instantiate objects.
I.e., give initial values to the fields in the newly-created instance.

At the level of the JVM, invoking a constructor

- sets aside memory for the instance and its fields,
- then executes the code in the constructor . . .
if the code in the constructor body begins with a call of `super`, then the corresponding constructor of the immediate superclass is executed;
Executing a Constructor

- if the code in the constructor body begins with a call of `super`, then the corresponding constructor of the immediate superclass is executed;
- if not, the compiler *adds* a call to a *default* constructor with no arguments: `super()`
... Executing a Constructor

- if the code in the constructor body begins with a call of `super`, then the corresponding constructor of the immediate superclass is executed;
- if not, the compiler adds a call to a default constructor with no arguments: `super()` — if there is no such constructor, the compiler reports an error.
E.g.

```java
class SuperTest {
    public SuperTest(int i){ }
}

class MySubclass extends SuperTest {
    public MySubclass(){ }
}
```

compiler output

file: SuperTest.java

SuperTest.java:8: cannot find symbol
symbol: constructor SuperTest()
location: class SuperTest
public MySubclass()
E.g.

```java
file: SuperTest.java

class SuperTest {
    public SuperTest(int i) {}  
}

class MySubclass extends SuperTest {
    public MySubclass() {}  
}

compiler output

SuperTest.java:8: cannot find symbol
  symbol:   constructor SuperTest()
  location: class SuperTest
          public MySubclass()
```
Java: Poor Design?

Generally, compilers doing things ‘behind the scenes’ can be confusing; the compile-time error above is especially confusing since the following is perfectly acceptable

```java
class Confusion {
    // no constructor
    public static void main(String[] args) {
        Confusion wtf = new Confusion();
    }
}
```
Generally, compilers doing things ‘behind the scenes’
can be confusing;
the compile-time error above is especially confusing
since the following is perfectly acceptable

class Confusion {
    // no constructor
    public static void main(String[] args) {
    }
}
Java: Poor Design?

Generally, compilers doing things ‘behind the scenes’ can be confusing; the compile-time error above is especially confusing since the following is perfectly acceptable

```java
file: Confusion.java

class Confusion {
    // no constructor
    public static void main(String[] args) {
        Confusion wtf = new Confusion();
    }
}
```
Summary

- Blocks, methods, classes and packages define scope
- Hide everything that doesn’t need to be visible
- super

Next:
CardHand