These slides lead you simply through OO Java, rarely use unexplained terms.
Examples, rather than formal definitions, are the norm.
Pages 2..3 are an index into the slides, helping you easily find what you want.
Many slides point to pages in the CS2110 text for more info.
Use the slides as a quick reference.
The ppt version, instead of the pdf version, is best, because you can do the Slide Show and see the animations, helping you to best read/understand each slide.

Type: Set of values together with operations on them

<table>
<thead>
<tr>
<th>Primitive types</th>
<th>byte</th>
<th>short</th>
<th>int</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer types</td>
<td>1 byte</td>
<td>2 bytes</td>
<td>4 bytes</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Real</td>
<td>float</td>
<td>double</td>
<td>~22.51E6</td>
<td>24.9</td>
</tr>
<tr>
<td>Character</td>
<td>char</td>
<td>2 bytes</td>
<td>'V'</td>
<td>'§'</td>
</tr>
<tr>
<td>Logical</td>
<td>boolean</td>
<td>true</td>
<td>false</td>
<td>and &amp; &amp; or &amp;&amp; not !</td>
</tr>
</tbody>
</table>

Index

abstract class 42–44
abstract method 44
access modifier 11
aliasing, 17
Array 50
constructor 10, 14, 24, 28
default 29
enums 81
equals function 37
exception 65–72
extend 27
Field 10, 12, 45
referencing 18
final 21
Function 10, 13
generic type 56
getter 13
immutable 46
Implements 60
Import 20
Indirect reference, 17
inherit 27
initializer 53
Interface 60
Junit testing 74–80
local variable 45
Method 10
narrower type 6, 35
calling 18

Matlab, Python weakly typed: A variable can contain any value —5, then “a string”, then an array, …
Java strongly typed: Must declare a variable with its type before you can use it. It can contain only values of that type
Type: Set of values together with operations on them

Type int:
values: -2147483648, -2147483647, ..., -3, -2, -1, 0, 1, 2, 3, 4, 5, ..., 2147483646, 2147483647
operations: +, -, *, /, %, unary –
b % c : remainder when b is divided by c. 67 % 60 = 7

Strong versus weak typing

Casting among types

Page A-9, inside back cover
Basic variable declaration

Declaration of a variable: gives name of variable, type of value it can contain

```java
int x;
```
Declares x, can contain an int value

```java
double area;
```
Declares area, can contain a double value

```java
int[] a;
```
Declares a, can contain a pointer to an int array. We explain arrays later

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>x</td>
</tr>
<tr>
<td>double</td>
<td>area</td>
</tr>
<tr>
<td>int[]</td>
<td>a</td>
</tr>
</tbody>
</table>
```

Assignment

```java
<variable> = <expression> ;
```
Type of <variable> must be same as or wider than type of <expression>

```java
x = area;
```
Illegal because type of x (int) is narrower than type of area (double)

```java
x = (int) area;
```
But you can cast the expression

```
<table>
<thead>
<tr>
<th>Type</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>x</td>
</tr>
<tr>
<td>double</td>
<td>area</td>
</tr>
</tbody>
</table>
```

Assignment of class Circle

```java
/** An instance (object) represents a circle */
public class Circle {
    private double radius; // radius of circle. radius >= 0

    public Circle(double radius) { ... } // Constructor
    public double getRadius() { ... } // Getter
    public void setRadius(double radius) { ... } // Setter
    public double area() { ... } // Area function

    // Put class declaration in file Circle.java
}
```

```
<table>
<thead>
<tr>
<th>Access modifier</th>
<th>Name of object</th>
<th>Address in memory</th>
<th>How we might write it on blackboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>Circle@ab14f324</td>
<td>radius 4.1</td>
<td>variable, called a field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getRadius() {}</td>
<td>functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>setRadius(double) {}</td>
<td>procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>area() {}</td>
<td>constructor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Circle(double)</td>
<td>we normally don't write body</td>
</tr>
</tbody>
</table>
```

Two aspects of a programming language

- Organization — structure
- Procedural — commands to do something

Example: Recipe book

- Organization: Several options; here is one:
  - Appetizers
  - list of recipes
  - Beverages
  - list of recipes
  - Soups
  - list of recipes
  - ...  
- Procedural: Recipe: sequence of instructions to carry out

```
<table>
<thead>
<tr>
<th>Two objects of class Circle</th>
<th>Two objects of class Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of object</td>
<td>Address in memory</td>
</tr>
<tr>
<td>Circle@ab14f324</td>
<td>radius 4.1</td>
</tr>
<tr>
<td>getRadius()</td>
<td>setRadius(double)</td>
</tr>
<tr>
<td>area()</td>
<td>Circle(double)</td>
</tr>
</tbody>
</table>
```

Declaration of class Circle

```java
public class Circle {
    public Circle(double radius) { ... } // Constructor
    public double getRadius() { ... } // Getter
    public void setRadius(double radius) { ... } // Setter
    public double area() { ... } // Area function

    // Put class declaration in file Circle.java
}
```

```
<table>
<thead>
<tr>
<th>Access modifier</th>
<th>Name of object</th>
<th>Address in memory</th>
<th>How we might write it on blackboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Circle@ab14f324</td>
<td>radius 4.1</td>
<td>variable, called a field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getRadius() {}</td>
<td>functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>setRadius(double) {}</td>
<td>procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>area() {}</td>
<td>constructor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Circle(double)</td>
<td>we normally don't write body</td>
</tr>
</tbody>
</table>
```

Declaration of field radius, in body of class Circle

```java
private double radius; // radius of circle. radius >= 0
```

```
<table>
<thead>
<tr>
<th>Access modifier</th>
<th>Name of object</th>
<th>Address in memory</th>
<th>How we might write it on blackboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>Circle@ab14f324</td>
<td>radius 4.1</td>
<td>variable, called a field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getRadius() {}</td>
<td>functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>setRadius(double) {}</td>
<td>procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>area() {}</td>
<td>constructor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Circle(double)</td>
<td>we normally don't write body</td>
</tr>
</tbody>
</table>
```

```java
public: Code everywhere can refer to Circle. Called access modifier
```

```
<table>
<thead>
<tr>
<th>Access modifier</th>
<th>Name of object</th>
<th>Address in memory</th>
<th>How we might write it on blackboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>Circle@ab14f324</td>
<td>radius 4.1</td>
<td>variable, called a field</td>
</tr>
<tr>
<td></td>
<td></td>
<td>getRadius() {}</td>
<td>functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>setRadius(double) {}</td>
<td>procedure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>area() {}</td>
<td>constructor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Circle(double)</td>
<td>we normally don't write body</td>
</tr>
</tbody>
</table>
```
Declaration of functions in class Circle

```java
/** return radius of this Circle */
public double getRadius() {
    return radius;
}

/** return area of Circle */
public double area() {
    return Math.PI*radius*radius;
}
```

Always specify method, saying precisely what it does.

Function header syntax: close to Python/Matlab, but return type `double` needed to say what type of value is returned.

Public so functions can be called from anywhere.

Called a getter: it gets value of a field.

Declaration of procedure in Circle

```java
/** Set radius to r. Precondition: r >= 0. */
public void setRadius(double r) {
    assert r >= 0;
    radius = r;
}
```

Always specify method, saying precisely what it does.

Function header syntax: close to Python/Matlab, but return type `double` needed to say what type of value is returned.

Public so functions can be called from anywhere.

Called a setter: it sets value in a field.

Declaration of constructor Circle

```java
/** Constructor: instance with radius r. Precondition: r >= 0 */
public Circle(double r) {
    assert r >= 0;
    radius = r;
}
```

A constructor is called when a new object is created (we show this soon).

Purpose of constructor: initialize fields of new object so that the class invariant is true.

```java
/** Constructor: instance with radius r. Precondition: r >= 0 */
public Circle(double r) {
    assert r >= 0;
    radius = r;
}
```

No constructor declared in a class? Java puts this one in, which does nothing, but very fast:

```java
public <class-name>() {}
```

Creating objects

New-expression: `new <constructor-call>`

Example: `new Circle(4.1)`

Evaluation is 3 steps:
1. Create new object of the given class, giving it a name. Fields have default values (e.g. 0 for `int`)
2. Execute `<constructor-call>` — in example, `Circle(4.1)`
3. Give as value of the expression the name of new object.

```java
Circle c;  // NEW
```

Consequences

1. Circle can be used as a type, with set of values: null and names of objects of class Circle
2. Objects are accessed indirectly. A variable of type Circle contains not the object but a pointer to it (i.e. its name)
3. More than one variable can contain the name of the same object. Called aliasing

```java
Circle d = c;
```

Referencing components of c

Suppose c and d contain the name Circle@ab14f324 — they contain pointers to the object.

If field radius is `public`, use `c.radius` to reference it

Examples: `c.radius = c.radius + 1; d.radius = c.radius + 3;`

Call function `area` using `c.area()` or `d.area()`

Call procedure `setRadius` to set the radius to 6 using `c.setRadius(6);` or `d.setRadius(6);`

```java
Call function area using c.area() or d.area()  
Call procedure setRadius to set the radius to 6 using c.setRadius(6); or d.setRadius(6);  
```
Value null
Value null denotes the absence of an object name or pointer

c = new Circle(0);
<= Circle@ab14f324
d = null;
<= null

c.area() has value 0.0

d.area() gives a "null-pointer exception" and program execution aborts (stops)

Static variables and methods
static: component does not go in objects. Only one copy of it

public class Circle { 
    final: PI can’t be changed. It’s a constant
    public static final double PI = 3.141592653589793;
    //** return area of c */
    public static double di(Circle c) { 
        return Math.PI * c.radius * c.radius;
    }
}

To use static PI and di:
Circle.PI
Circle.di(new Circle(5))

Use of this
public class Circle { 
    private double radius;
    //** Constr: instance with radius radius*/
    public Circle(double radius) { 
        radius = radius;
    }
    this evaluates to the name of the object in which is appears
    //** Constr: instance with radius radius*/
    public Circle(double radius) { 
        this.radius = radius;
    }
}

Avoid duplication: Call one constructor from other
Can save a lot if there are lots of fields

//** Constr: instance with horiz, vert side lengths sh, sv */
public Rectangle(double sh, double sv) { ...

//** Constr: instance with horiz, vert side lengths sh, sv */
public Rectangle(double sh, double sv) {
    sideH = sh; sideV = sv;
}

//** Constr: instance with side length s */
public Rectangle(double s) {
    sideH = s; sideV = s;
}

First alternative
Better alternative

Overloading
Possible to have two or more methods with same name

/** instance represents a rectangle */
public class Rectangle { 
    private double sideH, sideV; // Horiz, vert side lengths
    //** Constr: instance with horiz, vert side lengths sh, sv */
    public Rectangle(double sh, double sv) { ...
    }
    //** Constr: square with side length s */
    public Rectangle(double s) {
        sideH = s; sideV = s;
    }
}

Packages
package: set of related classes that appear in the same directory on your hard drive.

http://docs.oracle.com/javase/7/docs/api/
Contains specifications of all packages that come with Java. Use it often.

You will not write your own package right now, but you will use packages

Package java.io contains classes used for input/output. To be able to use these classes, put this statement before class declaration: import java.io.*;
* Means import all classes in package

Package java.lang does not need to be imported. Has many useful classes: Math, String, wrapper classes ...

Avoid duplication: Call one constructor from other
Can save a lot if there are lots of fields

//** Constr: instance with horiz, vert side lengths sh, sv */
public Rectangle(double sh, double sv) { ...

//** Constr: instance with side length s */
public Rectangle(double s) {
    sideH = s; sideV = s;
}

First alternative
Better alternative

/
Subclasses

Situation. We will have classes Circle, Rectangle, others: Circle: field radius: radius of circle

Want to place each object in the plane: A point (x, y) gives top-left of a rectangle or top-left of “bounding box” of a circle.

One way: add fields x and y to Circle, Rectangle, other classes for shapes. Not good: too much duplication of effort.

Better solution: use subclasses

Circle
radius
getRadius()
setRadius(double)
area()

Shape(…)  getX()  getY()

Circle is subclass of Shape
Shape is superclass of Circle

Put Circle components below (Circle is subclass)

Class Shape

/** An instance represents a shape at a point in the plane */
public class Shape {
    private double x, y;  // top-left point of bounding box
    public Shape (double x1, y1) {  
        x= x1;  y= y1;
    }  
    public double getX() {
        return x;
    }  
    public double getY() {
        return y;
    }
}

Circle
inherits all components of Shape: they are in objects of class Circle.

/** An instance represents circle at point in plane */
public class Circle extends Shape {
    all declarations as before except  
    /** Constructor: new Circle of radius r at (x, y)*/
    public Circle (double r, x, y) {  
        radius= r;
        super (x, y);
    }  
    getRadius()  setRadius(double)
    area()  Circle(double)
}

Rule. Constructor body must begin with call on another constructor.

If missing, Java inserts this: super();

Consequence: object always has a constructor, but it may not be one you want. In this case, error: Shape doesn’t have Shape().
Example of overriding: toString

Override an inherited method: define it in subclass

```java
/** return representation of this */
public String toString() {
    return "Circle radius \(5.3\) at \((20, 2)\)";
}
```

Put in class Shape
---

```
/** return representation of this */
public String toString() {
    return "Circle radius \(x + \), \(y + \)";
}
```

Put in class Circle
---

```
@Override
public String toString() {
    return c.toString();
}
```

super.toString()
---

Helps catch errors. Use it.

Example: toString() is special in Java

Good debugging tool: Define toString in every class you write, give values of (some of) fields of object.

```java
/** return representation of this */
public String toString() {
    return "Circle radius \(x + \), \(y + \)";
}
```

Put in class Shape

```
System.out.println("c is: " + c);
```

prints "c is (20, 2)"
---

Calling overridden method

Within method of class, use `super` to call overridden method — one in a higher partition, in some superclass.

```java
/** return representation of this */
public String toString() {
    return "Circle radius \(x + \), \(y + \)";
}
```

Put in class Circle

```
@Override
public String toString() {
    return c.toString();
}
```

Don’t need `@Override`.
---

Casting among class-types

(\(\text{int} \) \(5.0 / 3\)) // cast value of expression from double to int

```java
(\(\text{Shape}\) \(c\)) // cast value in \(c\) from Shape to \(\text{Circle}\)
```

Put in class Shape

```
Circle@x1
```

```
Object()
```

```
Object()
```

Explain, using this situation

```java
Circle c= new Circle(5.3, 2);
Shape d= (Shape) c;
```

Dont't need `@Override`.
---

Different perspectives of object

\(\text{Circle}\@x1\) is a \(\text{Circle}\), \(\text{Shape}\), \(\text{Object}\)

Cast \((\text{String})\ c\) is illegal because

\(\text{Circle}@x1\) is not a \(\text{String}\) — does not have a partition for \(\text{String}\)

```java
\(\text{Circle}@x1\)
```

```
\(\text{Object}\)
```

```
\(\text{Shape}\)
```

```
\(\text{Circle}\)
```

\(\text{Object}\) \(c\) widening cast, may be done automatically

\(\text{Circle}\) \(e\) narrowing cast, must be done explicitly

---

Page C-12

Page C-12

Page C-23, but not good

Page C-23, but not good

Page C-23, but not good

Page C-23, but not good
More on the perspective

b is an array of Shape objects
b[3] contains name of (pointer to) Shape object
b[3] has type Shape. Is b[3].area() legal?
NO. Have to do
((Trian) b[3]).area() 

NOT GOOD!!!

Shape s = (Shape) ob;
}
if public boolean
ob and this object at same point */
/** return true iff ob is a Shape and

shape has fields (x, y) to contain the position
of the shape in the plane. Each subclass describes
some enclosed kind of shape with an area
of the shape in the plane. Each subclass describes

E.g. overriding function equals (an automatic cast)
/** return true iff ob is a Shape and
ob and this object at same point */
public boolean equals(Object ob) {
 if (!(ob instanceof Shape)) {
 return false;
 }
 Shape s = (Shape) ob;
 return x == s.x && y == s.y;
}

Call d.equals(f)
Store arg f in parameter ob.
Automatic cast from C to Object
because ob has type Object

E.g. overriding function equals (instanceof)
/** return true iff ob is a Shape and
ob and this object at same point */
public boolean equals(Object ob) {
 if (!(ob instanceof Shape)) {
 return false;
 }

E.g. overriding function equals (need for cast)
/** return true iff ob is a Shape and
ob and this object at same point */
public boolean equals(Object ob) {
 if (!(ob instanceof Shape)) {
 return false;
 }
 Shape s = (Shape) ob;
 return x == s.x && y == s.y;
}

Need to test ob.x, ob.y — these are illegal! So cast ob to Shape. Then test

Motivating abstract classes
Shape has fields (x, y) to contain the position
of the shape in the plane. Each subclass describes
some enclosed kind of shape with an area

b[3].area() is illegal, even though each
Subclass object has function area()
Motivating abstract classes

Area() in class Shape doesn't return useful value

```java
public double area() { return 0.0; }
```

Problem: How to force subclasses to override area?

Problem: How to ban creation of Shape objects

```java
public abstract class Shape {
  public abstract double area();
}
```

Abstract class. Means can't create object of Shape:
```
new Shape(...)
```
syntactically illegal

Place abstract method only in abstract class.
Body is replaced by ;

Abstract method. Means it must be overridden in any subclass

Java has 4 kinds of variable

```java
public class Circle {
  private double radius;
  private static int t;
  public Circle(double r) {
    double r1 = r;
    radius = r1;
  }
}
```

Field: declared non-static. Is in every object of class. Default initial val depends on type, e.g. 0 for int

Class (static) var: declared static. Only one copy of it. Default initial val depends on type, e.g. 0 for int


Local variable: declared in method body. Created during call before exec. of body, discarded when call completed. No initial value. Scope: from declaration to end of block.

Why “wrapper” class?

Wrapper classes (for primitive types)
in package java.lang. Need no import

object of class Integer “wraps” one value of type int.
Object is immutable: can’t change its value.

Reasons for wrapper class Integer:
1. Allow treating an int value as an object.
2. Provide useful static variables, methods

```java
Integer k = new Integer(63);
int j = k.intValue();
```

Wrapper classes (for primitive types)

Wrapper class for each primitive type. Want to treat primitive value as an object? Just wrap it in an object of wrapper class!

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>Wrapper class</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>float</td>
<td>Float</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Wrapper class has:
• Instance methods, e.g. equals, constructors, toString,
• Useful static constants and methods.

```java
Integer k = new Integer(63);
int j = k.intValue();
```
Wrapper-class autoboxing in newer Java versions

**Autoboxing**: process of automatically creating a wrapper-class object to contain a primitive-type value. Java does it in many situations:

Instead of
\[
\text{Integer k = new Integer(63);}
\]
do
\[
\text{Integer k = 63;}
\]

**Auto-unboxing**: process of automatically extracting the value in a wrapper-class object. Java does it in many situations:

```java
// Extract the value from k, above:
Instead of
\[
\text{int i = k.intValue();}
\]
do
\[
\text{int i = k;}
\]
```

Array

Array: object. Can hold a fixed number of values of the same type. Array to right: 4 int values.

<table>
<thead>
<tr>
<th>Number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[]</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>int[]</td>
<td>3</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Arrays**

```java
// Array
int[] x;
null int[] Arrays

// Create array object of length 4, store its name in x
x = new int[4];

// Assign 2*x[0], i.e., -8, to x[3]
x[3] = 2 * x[0];

// Assign 5 to array element 2 and -4 to array element 0
x[2] = 5; x[0] = -4;

// x[2] is a reference to element number 2 of array x
```

**Array length**

Array length: an instance field of the array. This is why we write x.length, not x.length().

<table>
<thead>
<tr>
<th>Number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[1]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x[3]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Length field is final: cannot be changed. Length remains the same once the array has been created. We omit it in the rest of the pictures.

The length is not part of the array type. The type is int[]

An array variable can be assigned arrays of different lengths.

**Array initializers**

Instead of
\[
\text{int[]} c = \text{new int}[5];
\]

Use an array initializer:
\[
\text{int[]} c = \text{new int}[5];
\]

No expression between brackets [:].

**Ragged arrays: rows have different lengths**

```java
// Ragged arrays: rows have different lengths
int[][] b; // Declare variable b of type int[][]

b = \text{new int[2][]} // Create a 1-D array of length 2 and store its name in b. Its elements have type int[] (and start as null).
b[0]= \text{new int[1]} // Create int array, store its name in b[0].
b[1]= \text{new int[2]} // Create int array, store its name in b[1].
```
/** = first n rows of Pascal’s triangle. Precondition: 0 ≤ n */
public static int[][] pascalTriangle(int n) {
    int[][] b = new int[n][]; // array with n rows (can be 0!)
    // inv: rows 0..i-1 have been created
    for (int i = 0; i != b.length; i = i+1) {
        b[i] = new int[i+1]; // Create array for row i
        // Calculate row i of Pascal’s triangle
        b[i][0] = 1; // inv: b[i][0..j-1] have been created
        for (int j = 1; j < i; j = j+1) {
            b[i][j] = b[i-1][j-1] + b[i-1][j];
        }
        b[i][i] = 1;
    }
    return b;
}

Pascal’s Triangle in a ragged array
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

Basic class Box
private Object object;
public void set(Object ob) {
    object = ob;
}
public Object get() {
    return object;
}

New code
Box<Integer> b = new Box<Integer>();
b.set(new Integer(35));
Integer x = b.get();

Can extend only one class
public abstract class C1 {
    public abstract int m();
    public abstract int p();
}

Use abstract classes! Seems OK, because method bodies not given!
But Java does not allow this.
Instead, Java has a construct, the interface, which is like an abstract class.

Interface declaration and use of an interface
public class C implements C1, C2 {
    C must override all methods in C1 and C2
    ...
}

Field declared in interface automatically public, static, final
Must have initialization
Use of public, static, final optional

Methods declared in interface are automatically public, abstract
Use of public, abstract is optional
Use : not {...}
Casting with interfaces

```java
class B extends A implements C1, C2 {
    // ...
}
interface C1 {
    // ...
}
interface C2 {
    // ...
}
class A {
    // ...
}
b = new B();
```

What does object `b` look like?

Object `b` has 5 perspectives. Can cast `b` to any one of them at any time. Examples:

- `(C2)` `b`
- `(A)(C2)` `b`
- `(C1)` `(C2)` `b`

You’ll see such casting later.

Look at: interface java.lang.Comparable

```java
/** Comparable requires method compareTo */
public interface Comparable<T>
{
    /** = a negative integer if this object < c,
     = 0 if this object = c,
     = a positive integer if this object > c.
     Throw a ClassCastException if c cannot be cast to the class of this object. */
    int compareTo(T c);
}
```

Classes that implement Comparable

- Boolean
- Byte
- Double
- Integer
- ... String
- BigDecimal
- BigInteger
- Calendar
- Time
- Timestamp
- ...

We haven’t talked about Exceptions yet. Doesn’t matter here.

Note: Class `implements Comparable` decides what `<` and `>` mean!

**An instance maintains a time of day */

```java
class TimeOfDay implements Comparable<TimeOfDay> {
    int hour; // range 0..23
    int minute; // minute within the hour, in 0..59

    /** = -1 if this time less than ob’s time, 0 if same,
     1 if this time greater than ob’s time */
    public int compareTo(TimeOfDay ob) {
        Note TimeOfDay
        used here
        Note: Class
        implements
        Comparable
        if (hour < ob.hour) return -1;
        if (hour > ob.hour) return 1;
        if (minute < ob.minute) return -1;
        if (minute > ob.minute) return 1;
        return 0;
    }
    // Class has lots of other methods, not shown. Function
    compareTo allows us to compare objects, e.g. can use
    to sort an array of TimeOfDay objects.
}
```

**Sort array b, using selection sort */

```java
public static void sort(Comparable[] b) {
    // inv: b[0..i-1] sorted and contains smaller elements
    for (int i= 0; i < b.length; i= i+1) {
        // Store in j the position of smaller of b[i..]
        int j= i;
        // inv: b[j] is smallest of b[i..k-1]
        for (int k= i+1; k < b.length; k= k+1) {
            // b[k].compareTo(b[j]) < 0)  j= k;
            Comparable t= b[i]; b[i]= b[j]; b[j]= t;
        }
    }
}
```

```
TimeOfDay[] b;
... sort(b)

Beauty of interfaces: sorts an array C[] for any class C, as long as C implements interface Comparable.
```

Exceptions

```java
public static void main(String[] args) {
    int b= 3/0; // This is line 7
    This is line 7
    Division by 0 causes an “Exception to be thrown”.
    program stops with output:

    Exception in thread "main" java.lang.ArithmeticException: / by zero
    at C.main(C.java:7)

    The "Exception" that is “thrown”
    Happened in C.main on line 7
}
```

```java
 parsesInt throws a NumberFormatException if the arg is not an int (leading/trailing spaces OK)
```

```java
public static void main(String[] args) {
    int b= Integer.parseInt("3.2");
    This is line 7
    Used NFE instead of NumberFormatException to save space
    Output is:
    Exception in thread "main" java.lang.NFE: For input string: "3.2"
    at java.lang.NFE但对于Strings(NFE: java:48)
    at java.lang.Integer.parseInt(Integer.java:458)
    at java.lang.Integer.parseInt(Integer.java:499)
    at C.main(C.java:6)
    3.2 not an int
called from
called from
called from
Found error

See stack of calls that are not completed!
```
Exceptions and Errors
In package java.lang: class Throwable:

- **Throwable@x1**
  - detailMessage: "/ by zero"
  - getMessage()
  - Throwable()
  - Throwable(String)

When some kind of error occurs, an exception is "thrown" — you'll see what this means later.

An exception is an instance of class Throwable (or one of its subclasses)

Two constructors in class Throwable. Second one
stores its String parameter in field detailMessage.

So many different kind of exceptions that we have to organize them.

```
// ArithmeticException: / by zero
```

```
// RuntimeException
ArithmeticException
```

Do nothing with these
You can "handle" these

```
// Exception() Throwable()
// RuntimeException() ArithmeticException
```

```
// Exception(String)
// ArithmeticException(String)
```

```
// RunTimeException() ArithmeticException
// ArithmeticException() RunTimeException()
```

Subclass always has: 2 constructors, no fields, no other methods.
Constructor calls superclass constructor.

How to write an exception class

```java
/** An instance is an exception */
public class OurException extends Exception {
    /** Constructor: an instance with message m */
    public OurException(String m) {
        super(m);
    }
    /** Constructor: an instance with no message */
    public OurException() {
        super();
    }
}
```

The "throws" clause

```java
/** Class to illustrate exception handling */
public class Ex {
    public static void main(String...) {
        second();
    }
    public static void second() {
        third();
    }
    public static void third() {
        throw new OurException("I threw it");
    }
}
```

If Java asks for a throws clause, insert it. Otherwise, don’t be concerned with it.
Try statement: catching a thrown exception

```
try {
    statements
} catch (class-name e) {
    statements
}
```

Execution: Execute the try-block. Three cases arise: The try-block:
1. Does not throw an exception: End of execution.
2. Throws a class-name exception: execute the catch-block statements, with e containing the thrown exception.
3. Throws other exception: throw the object to the statement that called m.

JUnit testing class

A JUnit testing class is a class that contains procedures that are called to do “unit testing”. The units are generally methods in objects.

Eclipse has a simple way to create such a class:
1. In Package Explorer, select src directory for project
2. Use menu item File -> New -> JUnit Test Case
3. If the class you are testing is C, name the file Ctester

JUnit testing class looks like this:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CTester {
    @Test
    public void test() {
    }
}
```

To test a new class

To test a class, it is best to
1. Write a method a test procedure to test whether the constructor sets all fields properly, so that the class invariant is true. This will also test the getters. (see next slide)
2. Write a test procedure to test whether the setters do their job correctly.
3. Write a test procedure to test whether toString() is correct.
4. Write a separate method for each of the other constructors (if there are more)
5. Write other test procedures as is necessary to test other methods.
Testing setter methods

public class CTester {
    @Test
    public void testSetters() {
        C c1 = new C(5, 7);
        c1.setF1(6);
        assertEquals(6, c1.getF1());
        s2.setF2(-5);
        assertEquals(-5, c1.getF2());
    }
}

Assume C has 3 fields, f1, f2, and f3, with appropriate getter and setter methods.

Warning: don’t use static components

While it is possible to use fields or static variables in a JUnit test class, we advise against it at this point. You do not know when they are initialized (before the call of each test procedure, or once when you use Run ➔ Run, or once when class if first created, whatever).

Just use local variables where needed in a testing class.

Enums (or enumerations)

An enum: a class that lets you create mnemonic names for entities instead of having to use constants like 1, 2, 3, 4

The declaration below declares a class Suit.
After that, in any method, use Suit.Clubs, Suit.Diamonds, etc. as constants.

class Suit {Clubs, Diamonds, Hearts, Spades}

Testing for an enum constant

public enum Suit {Clubs, Diamonds, Hearts, Spades}

Suit s = Suit.Clubs;
Then
s == Suit.Clubs is true
s == Suit.Hearts is false

switch(s) {
case Clubs: color = "black"; break;
case Spades: color = "black"; break;
case Diamonds: color = "red"; break;
case Hearts: color = "red"; break;
}

Can use a switch statement

Type of s is Suit.
You cannot write Suit.Hearts instead of Hearts

Miscellaneous points about enums

This declaration is shorthand for a class that has a constructor, four constants (public static final variables), a static method, and some other components. Here are some points:

1. Suit is a subclass of Enum (in package java.lang)
2. It is not possible to create instances of class Suit, because its constructor is private!
3. It’s as if Clubs (as well as the other three names) is declared within class Suit as

   public static final Suit Clubs = new Suit(some values);

4. Static function values() returns a Suit[] containing the four constants. You can, for example, use it to print all of them:

   for (Suit s : Suit.values())
   System.out.println(s);

   Output:
   Clubs
   Diamonds
   Hearts
   Spades

5. Static function valueOf(String name) returns the enum constant with that name:

   Suit c = Suit.valueOf("Hearts");

   After the assignment, c contains (the name of) object Hearts
Miscellaneous points about enums

**public enum** Suit {Clubs, Diamonds, Hearts, Spades}

This declaration is shorthand for a class that has a constructor, four constants (public static final variables), a static method, and some other components. Here are some points:

6. Object Clubs (and the other three) has a function ordinal() that returns its position in the list

<table>
<thead>
<tr>
<th>Suit</th>
<th>ordinal()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td>0</td>
</tr>
<tr>
<td>Diamonds</td>
<td>1</td>
</tr>
</tbody>
</table>

We have only touched the surface of enums. E.g., in an enum declaration, you can write a private constructors, and instead of Clubs you can put a more elaborate structure. That's outside the scope of CS2110.