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.

**Index**

abstract class 41-43
abstract method 43
access modifier 10
array 49
initializer 52
length 50
ragged 53-54
assert 13
autoboxing 48
casting 6, 33, 60
catch clause 72
class dec 10
class invariant 11
Comparable 61
constructor 9, 14, 23, 27
default 28
equals function 36
exception 64-71
extend 26
field 9, 11, 44
function 9, 12
generic type 55
getter 12
immutable 45
import 19
inherit 26
initial 52
instanceof 39
interface 59
local variable 44
method 9
calling 17
narrower type 6, 54
new-expression 15
for array 51
null 18
Object (class) 29
overloading 21
overriding 30-31
package 19
parameter 13, 44
precondition 13
primitive type 5
private 11
procedure 9, 13
Throwable 66
throws clause 71
toString 30-32
try statement 72
try clause 72
Type: Set of values together with operations on them

**Strong versus weak typing**

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: Set of values together with operations on them

**Primitive types**

Integer types:
- byte 1 byte
- short 2 bytes
- int 4 bytes
- long 8 bytes

Real:
- float 4 bytes
- double 8 bytes

Character:
- char 2 bytes

Logical:
- boolean 1 bit

Inside back cover, A-6..7

**Casting among types**

(int) 3.2 casts double value 3.2 to an int

any number type

any number expression

narrow may be automatic cast wider

byte short int long float double

must be explicit cast, may truncate

char is a number type:

(int) 'Y'
(char) 86

Unicode representation: 86 'Y'

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;
```

Declaration of `x`, can contain an `int` value

```java
double area;
```

Declaration of `area`, can contain a `double` value

```java
int[] a;
```

Declaration of `a`, can contain a pointer to an `int` array. We explain arrays later

---

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

---

Two objects of class Circle

<table>
<thead>
<tr>
<th>Name of object</th>
<th>address in memory</th>
<th>How we might write it on blackboard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle@ab14324</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>variable, called a field</th>
<th>functions</th>
<th>procedure</th>
<th>constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>radius</td>
<td>getRadius() { ... }</td>
<td>setRadius(double) { ... }</td>
<td>Circle(double)</td>
</tr>
</tbody>
</table>

| Collection of field definitions and constraints is called the class invariant |

See B-1..10 | funcs, procs, constructors called methods |

---

Declaration of field radius, in body of class Circle

```
present double radius; // radius of circle. radius >= 0
```

Always put a definition of a field and constraints on it.
Collection of field definitions and constraints is called the class invariant

Access modifier `private`: can refer to radius only in code in Circle. Usually, fields are private

See B-5..6

---

Declaration of class Circle

```
/** An instance (object) represents a circle */
public class Circle {
  \*\* An instance (object) represents a circle \*/

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

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

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

  public static void main(String[] args) {
    Circle circle1 = new Circle(4.1);
    System.out.println(circle1.getRadius());
  }
}
```

Precede every class with a comment

Put declarations of fields, methods in class body:

```java
{ ... }
```

Put class declaration in file Circle.java

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

See B-5..6

---

Declaration of functions in class Circle

Called a getter: it gets value of a field

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

Execution of return expression: terminates execution of body and returns the value of the expression. The function call is done.

See B-6..10

---
### Declaration of procedure in Circle

It sets value in a field

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

Tells user not to call method with negative radius

Procedure: doesn’t return val.

Instead of return type, use void

### Declaration of constructor Circle

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: public <class-name>() {}
Packages

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

[http://docs.oracle.com/javase/7/docs/api/](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:

```java
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 …

Page B-25

Overloading

Possible to have two or more methods with same name

```java
/** 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) {
    sideH= sh; sideV= sv;
  }
  /** Constructor: square with side length s */
  public Rectangle(double s) {
    sideH= s; sideV= s;
  }
  ...
}
```

Page B-21

Avoid duplication: Call one constructor from other

Can save a lot if there are lots of fields

```java
/** Constr: instance with horiz, vert side lengths sh, sv */
public Rectangle(double sh, double sv) { … }
```

First alternative

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

Better alternative

```java
/** Constr: square with side length s */
public Rectangle(double s) {
  this (s, s);
}
```

Call on another constructor in same class: use this instead of class name

Page C-10

Static variables and methods

**static**: component does not go in objects. Only one copy of it

```java
public class Circle {
  public static final double PI= 3.141592653589793;
  public static double di(Circle c) {
    return Math.PI * c.radius * c.radius;
  }
}
```

Here’s PI and di

```java
Circle PI
Circle di(new Circle(5))
```

Page B-19..21

Use of this

```java
public class Circle {
  private double radius;
  /** Constr: instance with radius radius*/
  public Circle(double radius) {
    radius= radius;
  }
}
```

Doesn’t work because both occurrences of radius refer to parameter

```java
this
```

Evaluates to the name of the object in which it appears

```java
this.radius= radius;
```

Memorize this!

Page B-28

Subclasses

**Subclasses**

**Situation.** We will have classes Circle, Rectangle, others:

- Circle: field radius: radius of circle
- Rectangle: sideH, sideV: horizontal, vertical side lengths.

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**

```java
(1, 2)
```

Page C-5..14
Subclass and superclass

```java
/** An instance represents circle at point in plane */
public class Circle extends Shape {
    /** return value of "this object and ob"
     * @param ob other object
     * @return true or false
     */
    public boolean equals(Object ob) {
        // Implementation
        return x == ob.x && y == ob.y;
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }
}
```

Circle is subclass of Shape
Shape is superclass of Circle

Class Shape

Modify Circle constructor

```java
/** An instance represents circle at point in plane */
public class Circle extends Shape {
    /** Constructor: new Circle of radius r at (x, y)
     * @param r radius
     * @param x x-coordinate
     * @param y y-coordinate
     */
    public Circle(double r, double x, double y) {
        radius = r;
        x = x1; y = y1;
    }

    /** return x-coordinate of bounding box */
    public double getX() {
        return x;
    }

    /** return y-coordinate of bounding box */
    public double getY() {
        return y;
    }
}
```

Object: superset class of them all

Class doesn’t explicitly extend another one? It automatically extends class Object. Among other components, Object contains:

- Constructor: public Object() {}
- /** return name of object */
  public String toString()
  c.toString() is "Circle@x1"
- /** return value of "this object and ob are same". i.e. of this == ob */
  public boolean equals(Object ob)
  c.equals(ob) is true
  c.equals(new Circle(…)) is false

Override an inherited method: define it in subclass

```java
/** return representation of this */
public String toString() {
    // Implementation
    return "Circle@x1";
}
```
toString() is special in Java

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

```java
public String toString() {
    return "c is (" + x + " , " + y + ");
}
```

In some places where String is expected but class name appears, Java automatically calls toString.

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

Page B-17

Calling overridden method

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

```java
public @Override String toString() {
    return "Circle radius " +
    radius + ", " +
    super.toString();
}
```

c.toString() is
"Circle radius 5.3 at (20, 3)"

Page C-12

Casting among class-types

```java
Shape c = (Shape) e;
```

```
Circle@x1
```

Casting among class-types

```java
(int) (5.0 / 3)
// cast value of expression from double to int
```

(Shape) c // cast value in c from Circle to Shape

```
Circle@x1
```

Page C-23, but not good

Different perspectives of object

e looks at Circle@x1 from perspective of class Object, e.m(…) syntactically legal only if method m(…) is in Object partition.

Example: If e.toString() legal, e.getx() illegal.

d looks at Circle@x1 from perspective Of Shape.

d.m(…) syntactically legal only if m(…) is in Shape or Object partition.

Example: If e.area() illegal

Page C-23, not good

More on the perspective

b is an array of Shape objects

b[3] has type Shape. Is b[3].area() legal?

NO. Have to do

```java
((Trian)b[3]).area()
```

NOT GOOD!!!

Page C-23, but not good
More on the perspective
Better: Declare area() in class Shape

public double area() { return 0.0; }

Now, b[3].area() is syntactically legal
calls function area in partition Trian

E.g. overriding function equals (instanceof)
Spec says return false if ob not a Shape.
That’s what if-statement does
/** return true iff ob is a Shape and
the 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;
}

New operator: instanceof
if (ob instanceof Shape) { return true; }
else { return false; }

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

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[i].area() is illegal, even though each
Subclass object has function area()

Don’t want to cast down.
Instead, define area() in Shape

Motivating abstract classes
area() in class Shape doesn’t return useful value

public double area() { return 0.0; }

Problem: How to force subclasses to override area?
Problem: How to ban creation of Shape objects

...
Abstract class and method solves both problems

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

public abstract class Shape {

    public abstract double area();
    ...

}  

Abstract method. Means it must be overridden in any subclass

Java has 4 kinds of variable

public class Circle {
    private double radius;

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

}  

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

private static int t;  

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

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


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.

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

Wrapper classes (for primitive types)

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

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
Integer k = new Integer(63);
do
Integer k = 63;
This autoboxes the 63

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

Extract the value from k, above:
Instead of
int i = k.intValue();
do
int i = k;
This auto-unboxes value in k
**Array**

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

The type of the array:

```java
int[]
```

Variable contains name of the array.

```
@x
```

Basic form of a declaration:

```java
<type> <variable-name> ;
```

A declaration of x.

Does not create array, only declares x. x's initial value is null.

```
int[] x ;
```

Elements of array are numbered: 0, 1, 2, ..., x.length–1;

The type of the array:

```java
int[]
```

Variable contains name of the array.

```
x[]
```

Array length:

An instance field of the array.

This is why we write x.length, not x.length()

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.

```
int[] x;
```

The length is not part of the array type.

The type is int[].

An array variable can be assigned arrays of different lengths.

---

**Arrays**

```
int[] x ;
```

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

```
x[2]= 5; Assign 5 to array element 2
```

```
x[0]= -4; Assign -4 to array element 0
```

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

```
int k= 3;
x[k]= x[0]; Assign 2 * x[0], i.e. -8, to x[3]
x[k-1]= 6; Assign 6 to x[2]
```

---

**Array initializers**

Instead of

```java
```

Use an array initializer:

```java
int[] c = new int[5] {5, 4, 7, 6, 5};
```

---

**Ragged arrays: rows have different lengths**

```java
int[][] b; Declare variable b of type int[]
```

```
b= 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]= new int[] {17, 13, 19}; Create int array, store its name in b[0].
```

```
b[1]= new int[] {28, 95}; Create int array, store its name in b[1].
```

---

**Pascal’s Triangle in a ragged array**

```java
// 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!)
for (int i= 0; i != b.length; i= i+1) { // inv: rows 0...i-1 have been created
    b[i]= new int[i+1]; // Create array for row i
    // Calculate row i of Pascal’s triangle
    b[i][0]= 1;
    for (int j= 1; j < i; j= j+1) {
        b[i][j]= b[i][j-1] + b[i-1][j];
    }
}
return b;
```

---

**Array length**

Array length: an instance field of the array.

This is why we write x.length, not x.length().

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.
**Generic types** — made as simple as possible

Suppose you use `Box` to hold only `Integer` objects. When you get value out, you have to cast it to `Integer` to use it.

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

**Suppose you use** `Box` **to hold only** `Integer` **objects.** **When you get value out, you have to cast it to** `Integer` **to use it.**

**Generic types** — a way, when creating an object of class `Box`, to say that it will hold only `Integer` objects and avoid the need to cast.

```java
Box<Integer> b = new Box<Integer>();
b.set(new Integer(35));
Integer x = b.get();
```

**Basic class Box**

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

**Written using generic type**

```java
public class Box<T> {
    private T object;
    public void set(T ob) {
        object = ob;
    }
    public T get() {
        return object;
    }
}
```

**Can extend only one class**

```java
public class C extends C1, C2 {
    public void p() {
        ...; h = m(); ...
    }
}
```

If we allowed multiple inheritance, which `m` used?

```java
public interface C1 {
    int m();
    int p();
    int FF = 32;
}
```

```java
public interface C2 {
    int m();
    int q();
}
```

**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**

```java
public class C implements C1, C2 {
    ...
}
```

```java
public interface C1 {
    int m();
    int p();
    int FF = 32;
}
```

Field declared in interface automatically public, static, final

**Must have initialization**

**Use of public, static, final**

**Optional**

Eclipse: Create new interface? Create new class, change keyword `class` to `interface`

```java
class B extends A implements C1, C2 { ...
}
```

```java
interface C1 { ...

interface C2 { ...

class A { ...
```
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);
}
```

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

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`

`62`

`/** An instance maintains a time of day */`
`class TimeOfDay implements Comparable<TimeOfDay> {`

```java
int hour; // range 0..23
int minute; // minute within the hour, in 0..59
```

`Note: TimeOfDay used here`

```java
public int compareTo(TimeOfDay ob) {
  if (hour < ob.hour) return -1;
  if (hour > ob.hour) return 1;
  // {hour = ob.hour}
  if (minute < ob.minute) return -1;
  if (minute > ob.minute) return 1;
  return 0;
}
```

`Note TimeOfDay used here`

`63`

`/** Sort array b, using selection sort */
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) {
      if (b[k].compareTo(b[j]) < 0)  j= k;
    }
    Comparable t= b[i]; b[i]= b[j]; b[j]= t;
  }
}
```

`Exceptions

- public static void main(String[] args) {
-   int b= 3/0;
- }
```

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)
```

```
Happened in C.main on line 7
```

`Exceptions and Errors

In package java.lang: class Throwable:

```java
public static void main(String[] args) {
  int b= Integer.parseInt("3.2");
}
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "3.2"
```

```
Happened in C.main on line 7
```

Exceptions and Errors

In package java.lang: class Throwable:

```java
public static void main(String[] args) {
  int b= Integer.parseInt("3.2");
}
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "3.2"
```

```
Happened in C.main on line 7
```

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.
```

```
See stack of calls that are not completed!
```

```
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:forInputString(NFE.java:48)
at java.lang.Integer.parseInt(Integer.java:458)
```
Exceptions and Errors

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

- Throwable
- Exception
- RuntimeException
- ArithmeticException
- Error

组织领导这些异常。

- ArithmeticException: / by zero
- getMessage()
- detailMessage

Do nothing with these.

You can "handle" these.

Throwable() Throwable(String)
Exception() Exception(String)
RuntimeException() RuntimeException(String)
ArithmeticException() ArithmeticException(String)

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

Class: Call
Object a0 is thrown out to the call. Thrown to call of main: info printed
Ex.first();

Output
ArithmeticException: / by zero
at Ex.third(Ex.java:13)
at Ex.second(Ex.java:9)
at Ex.main(Ex.java:5)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Native Method)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.Method.invoke(Method.java:585)

Class: Call
Object a0 is thrown out to the call. Thrown to call of main: info printed
Ex.first();

Output
ArithmeticException: / by zero
at Ex.third(Ex.java:13)
at Ex.second(Ex.java:9)
at Ex.main(Ex.java:5)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Native Method)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:585)

Throw statement
Class: Call
Same thing, but with an explicit throw statement

Try statement: catching a thrown exception
try{
  statements
  catch (class-name e){
    catch-block
  }
}

Assume statement occurs in a method m

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 that is a subclass of Throwable: Assume statement occurs in a method m
3. Throws other exception: throw the object to the statement that called m.