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.

new-expression 16
for array 52
null 19
Object 10
creation 16
object name 10
Object (class) 30
overloading 22
overriding 31-32
package 20
parameter 14, 45
precondition 14
primitive type 5
private 12
procedure 10, 14

public 11
ragged array 54-55
return statement 13
return type 13
setter 14
shadowing 31
static 21, 45
strongly typed 4
subclass 25
super 28, 33
superclass 27
this 23, 24
throw stmt 70

Throws 67
throws clause 72
toString 31-33
try statement 73
try clause 73
type 4
generic 56-57
variable decl 7
void 14
weakly typed 4
wider type 6, 35
wrapper class 46

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

Primitive types
Integer types:
byte 1 byte
short 2 bytes
int 4 bytes
long 8 bytes
usual operators
Real:
float 4 bytes
double 8 bytes
~2.51E6
24.9
usual operators
Character:
char 2 bytes
‘V’ ‘$’ ‘n’
no operators
Logical:
boolean 1 bit
true
false
and & or !
Single quote

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) ‘V’
(char) 86
Unicode representation: 86 ‘V’

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

Assignment

```java`
<variable> = <expression> ;
```

Type of <variable> must be same as or wider than type of <expression>

Run-time type checking prevents illegal assignment.

```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

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

Variable, called a field

Name of object

Address in memory

How we might write it on blackboard

Methods

Functions

Procedures

Constructors

We normally don’t write body

Put declarations of fields, methods in class body:

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

Put class declaration in file Circle.java

Always put a definition of a field and constraints on it.

Collection of field definitions and constraints is called the class invariant

Declarations of field radius, in body of class Circle

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

Page A-6

Page A-6

Page A-6

Page A-6

Page B-5

Page B-5

Page B-5

Page B-5..6

Page B-5..6

Page 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;
}

/** 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.

Declaration of procedure in Circle

Called a setter: It sets value in a field

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

Procedure: doesn’t return value. Instead of return type, use void.

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.

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.

/** 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 <classname>() {}

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

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);

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.

Circles are accessed indirectly. A variable of type Circle contains not the object but a pointer to it (i.e. its name). More than one variable can contain the name of the same object. Called aliasing.
**Value null**

Value **null** denotes the absence of an object name or pointer.

c = new Circle(0);

Circle@ab14f324

d= null;

d = null

c.area() has value 0.0

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

**Packages**

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

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 ...

**Static variables and methods**

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

public class Circle {
    static final double PI = 3.141592653589793;
    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))

**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;
    }
}
```

Lists of parameter types must differ in some way

**Avoid duplication: Call one constructor from other**

Can save a lot if there are lots of fields

```java
/** Constr: instance with horiz, vert sidelengths sh, sv */
public Rectangle(double sh, double sv) {
    this(sh, s);
}
```

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
}
```

this(...) must be first statement in constructor body

**Use of this**

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

This works

Memorize this!

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

this evaluates to the name of the object in which it appears

First alternative

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

This works

Memorize this!

First 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

```java
/** An instance represents a shape at a point in the plane */
public class Shape {
    private double x, y; // top-left point of bounding box

    /** Constructor: a Shape at point (x1, y1) */
    public Shape (double x1, double y1) {
        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;
    }
}
```

Class Shape

```
/** An instance represents circle at point in plane */
public class Circle extends Shape {
    all declarations as before

    Circle is subclass of Shape
    Shape is superclass of Circle

    /** Constructor: new Circle of radius r at (x, y)*/
    public Circle (double r, double x, double y) {
        super (x, y);
        radius= r;
    }

    /** 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(d) is true
        c.equals(new Circle(...)) is false
```

Modify Circle constructor

```
/** 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, double x, double y) {
        super (x, y);
        radius= r;
    }

    /** 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:

```
/** An instance represents a shape at a point in the plane */
public class Shape {
    private double x, y; // top-left point of bounding box

    /** Constructor: a Shape at point (x1, y1) */
    public Shape (double x1, double y1) {
        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;
    }

    /** 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(d) is true
        c.equals(new Circle(...)) is false
```

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() {
    System.out.println("c is: " + c);
    System.out.println("c is (20, 2)" + c.toString());
    return "Circle radius 5.3 at (20, 3)";
}
```

---

### 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() {
    System.out.println("c is (20, 2)" + c.toString());
    return "Circle radius 5.3 at (20, 3)";
}
```

---

### Casting among class-types

**Important:** Object Circle@x1 has partitions for Object, Shape, Circle. Can be cast only to these three classes.

- Circle@x1 is a Circle, Shape, Object.
- Cast (String) c is illegal because Circle@x1 is not a String — does not have a partition for String

```java
public Circle(…) 
getX() 
getY()
```

---

### 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: 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: e.area() illegal

```java
public Circle(…) 
setRadius(double)
getRadius()
```

---

### 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 "(20, 2)" + c.toString();
}
```
b is an array of Shape objects
b[i] 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!!!

E.g. overriding function equals (an automatic cast)

```java
/** 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;
}
```

// Store arg f in parameter ob. Automatic cast from C to Object because ob has type Object
b[3].area() is syntactically legal

E.g. overriding function equals (instanceof)

```java
/** 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;
    }
    // New operator: instanceof
    if (ob instanceof C) {
        Object c = (C) ob;
        // C@@???
        // Object
        // Shape
        // This is syntactically legal
    }
}
```

```
E.g. overriding function equals (need for cast)

```java
/** 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;
}
```

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()
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 Shape

public double area();

Circle@x

... Object

... Shape

... Circle

area()

... Shape

... Circle

area()

... Shape

... Circle

area()

Trian@z

... Object

... Shape

... Trian

area()

... Shape

... Trian

area()

Rect@y

... Object

... Shape

... Rect

area()

... Shape

... Rect

area()

0   1   2   3   4    …

b

Shape[]

Java has 4 kinds of variable

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.

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.

Integer.MIN_VALUE:
smallest int value: ~2^31

Static components:
MIN_VALUE
toString(int) toBinary(int) parsedInt(String)

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!

Primitive type   Wrapper class
int             Integer
long            Long
float           Float
double          Double
char            Character
boolean         Boolean

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

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:

```java
Instead of Integer k = new Integer(63);
do 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 int i = k.intValue();
do int i = 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
[ ]
```

Variable contains name of the array. `x[]` integer

Basic form of a declaration:

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

A declaration of x.

```java
int[] x ;
```

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

Array length

Array length: an instance field of the array.

```java
x.length
```

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.

```java
length
```

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

```java
int[] c = new int[5];
```

Use an array initializer:

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

No expression between brackets `[ ]`.

Array initializers: gives values to be in the array initially. Values must have the same type, in this case, int. Length of array is number of values in the list.

Ragged arrays: rows have different lengths

```java
int[][] b;  
```

Declare variable b of type int[]

```java
b = new int[2][2] ;
```

Create a 1-D array of length 2 and store its name in b. Its elements have type int[] (and start as null).

```java
b[0]= new int[] {17, 13, 19};
```

Create int array, store its name in b[0].

```java
b[1]= new int[] {28, 95};
```

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++) {
        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++) {
            b[i][j] = b[i-1][j-1] + b[i-1][j];
        }
        b[i][i] = 1;
    }
    return b;
}

Can extend only one class
public class C extends C2 { ...
    ...
    Field declared in interface are automatically public, static, final
    Must have initialization
    Use of public, static, final optional
}

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

Add C1, C2 as new dimensions:

```
C2  
C1
```

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

(C2) b                  (Object) b
(A)(C2) b          (C1) (C2) b
```

You’ll see such casting later.

Look at:

```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
- …

Note: Class implements Comparable.

```java
/** An instance maintains a time of day */  
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) {  
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;
}  
```

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 */  
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..k-1]
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;
}
```

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;
}
```

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:5)
Happened in C.main on line 7  
The "Exception" that is “thrown”
```

```java
parseInt 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");
}
```

Output is:

```
Found error on line 48
```

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

```
```

Note: Class implements Comparable.

```
```
Exceptions and Errors

In package java.lang: class Throwable:

- 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
RuntimeException

Do nothing with these
You can "handle" these
Subclass always has: 2 constructors, no fields, no other methods.
Constructor calls superclass constructor.

Creating and throwing and Exception

Class: Ex
Call
Object a0 is thrown out to the call.
Thrown to call of main: info printed
Output
ArithmeticException: / by zero
at Ex.third(Ex.java:13)
at Ex.second(Ex.java:9)
at Ex.main(Ex.java:5)

Throw statement
Class: Ex
Call
Same thing, but with an explicit throw statement
Output
ArithmeticException: / by zero
at Ex.third(Ex.java:13)
at Ex.second(Ex.java:9)
at Ex.main(Ex.java:5)

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() throws OurException {
        second();
    }
    public static void second() throws OurException {
        third();
    }
    public static void third() throws OurException {
        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.

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:

```java
import static org.junit.Assert.*;
import org.junit.Test;
public class CTester {
    @Test
    public void test() {
    }
}
```

Put as many different test() method, with mnemonically chosen names.

To call all such methods, select file CTester in the Package Explorer and then use menu item Run -> Run

What to put in a test method

```java
public void testFail() {
    fail("Not yet implemented");
}
```

Causes execution of method call to abort with a message

Testing 2 calls on static method m of C.
Put in as many tests as you need

```java
assertEquals(expected value, computed value);
```

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 a constructor

```java
public void testConstructor() {
    C c1 = new C(5, 7);
    assertEquals(5, c1.getF1());
    assertEquals(7, c1.getF2());
    assertEquals(20, c1.getF3());
}
```

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

Assume the 5 is for f1, the 7 is for f2, and f3 is to be initialized to 20.
This code creates a new objects and tests whether all fields are properly set.
Testing setter methods

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

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

Testing for an enum constant

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

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

Can use a switch statement

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

Type of s is Suit.

```
You cannot write Suit.Hearts instead of 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:

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);
```

You don’t care what 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 it position in the list

Suit.Clubs.ordinal() is 0
Suit.Diamonds.ordinal() is 1

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