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.

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

<table>
<thead>
<tr>
<th>Primitive types</th>
<th>Integer types:</th>
<th>byte</th>
<th>short</th>
<th>int</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 byte</td>
<td>2 bytes</td>
<td>4 bytes</td>
<td>8 bytes</td>
<td></td>
</tr>
<tr>
<td>Real:</td>
<td>float</td>
<td>double 4 bytes</td>
<td>8 bytes</td>
<td>-22.51E6</td>
<td>24.9</td>
</tr>
<tr>
<td>Character:</td>
<td>char</td>
<td>2 bytes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logical:</td>
<td>boolean</td>
<td>true false</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Casting among types

(char) is a number type: (int) 3.2 casts double value 3.2 to an int

\[
b \% c : \text{remainder when } b \text{ is divided by } c \quad 67 \% 60 = 7
\]
**Basic variable declaration**

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

- `int x;` Declaration of `x`, can contain an `int` value
- `double area;` Declaration of `area`, can contain a `double` value
- `int[] a;` Declaration of `a`, can contain a pointer to an `int` array. We explain arrays later

```
x 5  int area  20.1 double a
```

---

**Assignment**

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

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

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

```
x = (int) area;
```

But you can cast the expression

```
x = area;
```

---

**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

---

**Declaration of class Circle**

```
/** An instance (object) represents a circle */
public class Circle {
    // Precede every class with a comment
    Put declarations of fields, methods in class body: {
        // ... }
    Put class declaration in file Circle.java
}

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

---

**Declaration of field radius, in body of class Circle**

```
private 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**

---

**Two objects of class Circle**

- Name of object: `Circle@ab14f324`
- Address in memory: `Circle@x1`
- How we might write it on blackboard

- Variable, called a field
- Functions
- Procedure
- Constructor
- We normally don’t write body

See B-1..10

- funcs, procs, constructors called **methods**

---

**Multi-line comment starts with /* ends with */**

```
/** An instance (object) represents a circle */
```

---

**One-line comment starts with // ends at end of line**

```
// An instance (object) represents a circle
```

---

**Put declarations of fields, methods in class body: {**

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

---

**Page A-6**

---

**Page B-5**

---

**Page B-5..6**

---

**Page B-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

Public so functions can be called from anywhere
Page B-6..10

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;
}

Tells user not to call method with negative radius
Function: doesn't return val. Instead of return type, use void
Procedure: doesn't return val.

Called a getter:
It gets value of a field
Called a setter:
It sets value in a field
Page B-15..16

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.

Circle c; c= new Circle(4.1);
Circle c= new Circle(4.1);
Circle c= new Circle(4.1);

No constructor declared in a class? Java puts this one in, which does nothing, but very fast:
public <class-name>() {}

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

Example: Execute Circle d= c; and variables d and c contain the same value.
Circle c= ab14f324
Circle d= ab14f324

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()
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).

Static variables and methods

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

public class Circle {
  private double radius;

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

  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))

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) {
    sideH = sh;  sideV = sv;
  }

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

  \ 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

/** Constr: instance with horiz, vert sidelengths sh, sv */
public Rectangle(double sh, double sv) {
  \ First alternative...
}

First alterative

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

Better alternative

/** 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

public class Circle {
  private double radius;

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

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

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

  this works
}

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.

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 …

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

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

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circle</td>
<td>field <code>radius</code>: radius of circle</td>
</tr>
<tr>
<td>Rectangle</td>
<td><code>sideH</code>, <code>sideV</code>: horizontal, vertical side lengths.</td>
</tr>
</tbody>
</table>

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

```
(1, 2)  (20, 2)
```

[Diagram of Circle and Rectangle with fields and methods]

```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
    // ... other methods and fields...

    /** Constructor: 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`

```
Circle@x1
```

```java
/** An instance represents circle at point in plane */
public class Circle extends Shape {
    all declarations as before except
    Circle is subclass of Shape
    Shape is superclass of Circle
    Circle@x1

    x 20  y 2  Shape
    Shape(...)
    getRadius()
    getRadius(double)
    setRadius(double)
    area()  Circle(double)

    radius = 5.3
    getRadius()
    setRadius(double)
    area()  Circle(double)

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

    /* Principle: initialize superclass fields first, then subclass fields. */
    Implementation: Start constructor with call on superclass constructor
```

Page C-9

```
Modify Circle constructor
```

```java
/** An instance represents circle at point in plane */
public class Circle extends Shape {
    all declarations as before except
    Circle inherits all components of Shape: they are in objects of class Circle.

    put Shape components above
    put Circle components below (Circle is subclass)

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

    // ... other methods and fields...

    Circle@x1
    x 20  y 2  Shape
    Shape(...)
    getRadius()
    getRadius(double)
    setRadius(double)
    area()  Circle(double)

    radius = 5.3
    getRadius()
    setRadius(double)
    area()  Circle(double)

    // ... other methods and fields...

    /* Principle: initialize superclass fields first, then subclass fields. */
    Implementation: Start constructor with call on superclass constructor
```

Page C-9

```
Object: superest class of them all
```

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

- Constructor: `public Object()` 
- `toString()` method
- `equals(Object)` method

```
/** return name of object */
public String toString() {
    return this.toString();
}

/** return value of “this object and ob are same”, i.e. of this == ob */
public boolean equals(Object ob) {
    if (this == ob) return true;
    c.equals(d) is true
    c.equals(new Circle(...)) is false
```

Page C-18
Example of overriding: toString

**Override an inherited method:** define it in subclass

```java
public String toString() {
    return "Circle radius 5.3 at (20, 3)";
}
```

Put in class Shape.

**toString() is special in Java**

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

```java
public String toString() {
    return "Circle radius 5.3 at (20, 3)";
}
```

Put in class Shape.

Calling overridden method

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

```java
public String toString() {
    return "Circle radius 5.3 at (20, 3)"
        + super.toString();
}
```

Example:

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

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

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

Casting among class-types

```java
// cast value of expression from double to int
(Shape) c
```

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
```

Casting among class-types

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

```
Cast String c is illegal because Circle@x1 is not a String — does not have a partition for String
```

Casting among class-types

```java
// cast value in c from Object to String
String toString()
```

```
Put in class Circle
```

Important:

```java
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

```
Object c widen cast, may be done automatically
(Circle) e narrowing cast, must be done explicitly
```

```
Class casting: costs nothing at runtime, just provides different perspective on object.
```

```
Type of variable
```

```
Shape(…)  getX()  getY(
```

```
area()  Circle(…)  setRadius(
```

```
radius
```

```
Circle
```

```
Object
```

```
Object
```

Page C-23, but not good

Page C-12
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!!!

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;
}
```

Call d.equals(f)

Store arg f in parameter ob. Automatic cast from C to Object because ob has type Object

```java
Object d = new Circle(3.0); Shape f = d;
```

E.g. overriding function equals (instanceof)

```java
public boolean equals(Object ob) {
 if (!(ob instanceof Shape)) {
 return false;
 } ...
}
```

Now, b[3].area() 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 == ob.y;
}
```

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

```java
s = new Circle(3.0); ob = s;
```

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

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

Problem: How to force subclasses to override area?

Problem: How to ban creation of Shape objects

```java
class Circle extends Shape{
  double radius;
  public Circle(double r){
    this.radius = r;
  }
}
```

```java
class Rect extends Shape{
  double width, height;
  public Rect(double w, double h){
    this.width = w;
    this.height = h;
  }
}
```

```java
class Triang extends Shape{
  double base, height;
  public Triang(double b, double h){
    this.base = b;
    this.height = h;
  }
}
```

```java
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

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

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

Place abstract method only in abstract class.
Body is replaced by :
```
```

```
```

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

```
```

```
```

Why “wrapper” class?

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

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

```java
Wrapper class for each primitive type
```

<table>
<thead>
<tr>
<th>Primitive type</th>
<th>Wrapper class</th>
<th>Description</th>
</tr>
</thead>
</table>
| int            | Integer       | 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();
```

Page A-51..54
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:
```
int[]
```

Variable contains name of the array. `x [ ]@x3 int[]`

Basic form of a declaration:
```
<type> <variable-name>
```

A declaration of `x`.
```
int[] x;
```

Elements of array are numbered: 0, 1, 2, ..., `x.length–1`;
```
int[] x;
```

Array length

Array length: an instance field of the array.
```
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.
```
x.length
```

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

An array variable can be assigned arrays of different lengths.
```
int[] x;
```

Array initializers

Instead of
```
int[] c = new int[5];
```

Use an array initializer:
```
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.
```
int[] c = new int[] {5, 4, 7, 6, 5};
```

Arrays

```
x = new int[4];
```

Create array object of length 4, store its name in `x`.
```
x = new int[ ] {5, 4, 7, 6, 5};
```

Create an array initializer:
```
int[] c = new int[] {5, 4, 7, 6, 5};
```

Ragged arrays: rows have different lengths

```
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]`.  

```
b[0][ ]
```

Create `int` array, store its name in `b[0][ ]`.
```
b[1][ ]
```

Create `int` array, store its name in `b[1][ ]`.

```
b[0][ ]
```

Create `int` array, store its name in `b[0][ ]`.
```
b[1][ ]
```

Create `int` array, store its name in `b[1][ ]`.
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
        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
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

Can extend only one class

public class C extends C2 { ...

public abstract class C1 {
    public abstract int m();
    public abstract int p();
}

public abstract class C2 {
    public abstract int m();
    public abstract 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.

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.

Box b = new Box<>();
Object x = b.get();

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

Box b = new Box<>(new Integer(35));
Object x = b.get();

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

Basic class Box

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

parameter T (you choose name)

Basic class Box

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

Written using generic type

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

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

```java
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: 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.

```java
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;
}
```

Division by 0 causes an “Exception to be thrown”. Program stops with output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

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

Output is:

```
Exception in thread "main" java.lang.ParseException: For input string: "3.2"
at java.lang.ParseException\nforInputString\n(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
```

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

- Two constructors in class Throwable. Second one stores its String parameter in field detailMessage.
- An exception is an instance of class Throwable (or one of its subclasses).

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

### Creating and throwing an Exception

```java
public class Ex {
    public static void main(...) {
        second();
    }
    public static void second() throws OurException {
        third();
    }
    public static void third() throws OurException {
        throw new OurException("mine");
    }
}
```

Object a0 is thrown out to the call. Thrown to call of main: info printed

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(…)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(…)
at java.lang.reflect.Method.invoke(Method.java:585)

### 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();
    }
}
```

**Class to illustrate exception handling**

```java
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("mine");
    }
}
```

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
}
```

Assume statement occurs in a method \( m \)

class-name that is a subclass of Throwable

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 \( \rightarrow \) New \( \rightarrow \) Junit Test Case
3. If the class you are texting 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 \( \rightarrow \) Run
```

What to put in a test method

```java
... public class CTester {
    @Test
    public void testFail() {
        fail("Not yet implemented");
    }
    @Test
    public void testM() {
        assertEquals(5, C.m(30));
        assertEquals(20, C.m(0));
    }
    assertEquals(expected value, computed value);
}
```

Testing a constructor

```java
... public class CTester {
    @Test
    public void testConstructor() {
        C c1 = new C(5, 7);
        assertEquals(5, c1.getF1());
        assertEquals(7, c1.getF2());
        assertEquals(20, c1.getF3());
    }
    Note: purpose of procedure is to test constructor, but the method also tests the getter methods.
    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.
```

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.

```java
To call all such methods, select file CTester in the Package Explorer and then use menu item Run \( \rightarrow \) Run
```
Testing setter methods

```java
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.

```java
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

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

```java
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

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

You don’t care what values

Output:

<table>
<thead>
<tr>
<th>Clubs</th>
<th>Diamonds</th>
<th>Hearts</th>
<th>Spades</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clubs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>You can see that toString in object Clubs returns the string “Clubs”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

```java
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:

   - 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.