

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 42-44	Comparable 63	getter 13
abstract method 44	Constructor 10, 14, 24, 28	immutable 46
access modifier 11	default 29	Implements 60
aliasing, 17	enums 81	Import 20
Array 50	equals function 37	Indirect reference, 17
initializer 53	exception 65-72	inherit 27
length 51	extend 27	initializer 53
ragged 54-55	Field 10, 12, 45	Instanceof 40
assert 14	referencing 18	Interface 60
assignment 8	final 21	JUnit testing 74-80
autoboxing 49	Function 10, 13	local variable 45
casting 6, 34, 61	generic type 56	Method 10
catch clause 73		calling 18
class decl 11		narrower type 6, 35
class invariant 12		

Index

new-expression 16	public 11	Throwable 67
for array 52	ragged array 54-55	throws clause 72
null 19	return statement 13	toString 31-33
Object 10	return type 13	try statement 73
creation 16	setter 14	try clause 73
object name 10	shadowing 31	type 4
Object (class) 30	static 21, 45	generic 56-57
overloading 22	strongly typed 4	variable decl 7
overriding 31-32	subclass 25	void 14
package 20	super 28, 33	weakly typed 4
parameter 14, 45	superclass 27	wider type 6, 35
precondition 14	this 23, 24	wrapper class 46
primitive type 5	throw stmt 70	
private 12		
procedure 10, 14		

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 **int:** $-2^{31} .. 2^{31}-1$

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. $67 \% 60 = 7$

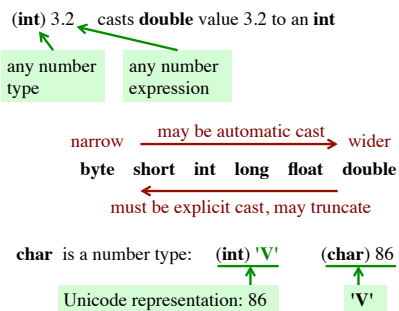
Type: Set of values together with operations on them

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	$-22.51E6$ 24.9		usual operators
Character:	char 2 bytes	'V'	'\$'	'\n'	no operators
Logical:	boolean 1 bit	true	false		and && or not !

Inside back cover, A-6..7

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

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 **int[]**

Page A-6 7

Assignment

`<variable> = <expression>;`

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

~~x = area;~~ Illegal because type of x (**int**) is narrower than type of area (**double**)

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

x 5 **int** area 20.0 **double**

Page A-6 8

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

structural
objects
classes
interface
inheritance

procedural
assignment
return
if-statement
iteration (loops)
function call
recursion

miscellaneous
GUIs
exception handling
Testing/debugging

11

Two objects of class Circle

Name of object: Circle@ab14f324

address in memory: 4.1

radius: 4.1

getRadius() { ... }

setRadius(**double**) { ... }

area() { ... }

Circle(**double**) { ... }

How we might write it on blackboard: Circle@x1

radius: 5.3

getRadius()

setRadius(**double**)

area()

Circle(**double**)

variable, called a **field** functions procedure constructor we normally don't write body

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

12

Declaration of class Circle

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

`/** An instance (object) represents a circle */` Precede every class with a comment

```
public class Circle {
    // 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

Page B-5 11

Declaration of field radius, in body of class Circle

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

```
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**

Page B-5..6 12

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

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

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

Procedure: doesn't return val. Instead of return type, use **void**

Declaration of parameter r. Parameter: var declared within () of a method header

The call **setRadius(-1)**; falsifies class invariant because **radius** should be ≥ 0 . User's fault! Precondition told user not to do it. Make method better by putting in **assert** statement.

Execution of **assert e**; aborts program with error message if **boolean** expression **e** is false.

Page B-6..10

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;
}

```

Constructor:

1. no return type
2. no **void**
3. Name of constructor is name of class

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

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);

```

Evaluate new expression:

1. Create object
2. Execute constructor call
3. Value of exp: **Circle@ab14f324**

Finish assignment

Page B-3

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.

```

c Circle@ab14f324
d Circle@ab14f324

```

```

Circle@ab14f324
radius 0.0
getRadius() { ... }
setRadius(double) { ... }
area() { ... }
Circle(double) { ... }

```

17

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);**

```

Circle@ab14f324
radius 0.0
getRadius() { ... }
setRadius(double) { ... }
area() { ... }
Circle(double) { ... }

```

```

c Circle@ab14f324 d Circle@ab14f324

```

18

Value null

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

```
c = new Circle(0);
d = null;
```

Diagram: Variable **c** points to object **Circle@ab14f324**. Variable **d** points to **null**.

c.area() has value 0.0

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

```
Circle@ab14f324
radius 0.0
getRadius() { ... }
setRadius(double) { ... }
diameter() { ... }
Circle(double) { ... }
```

19

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 ...

Page B-25

20

Static variables and methods

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

```
public class Circle {
    declarations as before
    public static final double PI = 3.141592653589793;
    /** return area of c */
    public static double di(Circle c) {
        return Math.PI * c.radius * c.radius;
    }
}
```

Diagram: **final: PI can't be changed It's a constant**. **Here's PI and di**. **PI 3.1415...**. **di(Circle) {...}**

To use static PI and di:

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

Circle@x1
Components as before, but not PI, di

Circle@x2
Components as before, but not PI, di

Page B-19..21

21

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

Diagram: **Lists of parameter types must differ in some way**

Page B-21

22

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

Diagram: **Doesn't work because both occurrences of radius refer to parameter**. **Memorize this!**. **This works**

Page B-28

23

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

Diagram: **First alternative**. **Better alternative**. **Call on another constructor in same class: use this instead of class name**. **this(...) must be first statement in constructor body**

Page C-10

24

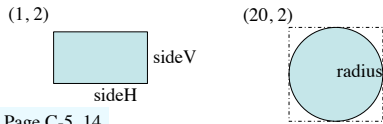
Subclasses

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



Page C-5..14

25

```

/** 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

26

Subclass and superclass

Circle is subclass of **Shape**
Shape is superclass of **Circle**

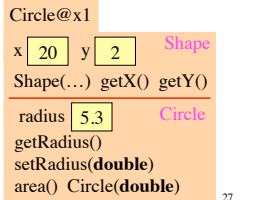
```

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

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

put **Shape** components above

put **Circle** components below (**Circle** is subclass)



27

Modify Circle constructor

Circle is subclass of **Shape**
Shape is superclass of **Circle**

Circle is subclass of **Shape**
Shape is superclass of **Circle**

Circle is subclass of **Shape**
Shape is superclass of **Circle**

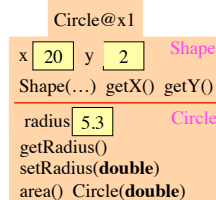
```

/** 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, double x, double y) {
        super(x, y); // how to call constructor in superclass
        radius = r;
    }
}
    
```

Principle: initialize superclass fields first, then subclass fields.

Implementation: Start constructor with call on superclass constructor

Page C-9



Default Constructor Call

Circle is subclass of **Shape**
Shape is superclass of **Circle**

Circle is subclass of **Shape**
Shape is superclass of **Circle**

Circle is subclass of **Shape**
Shape is superclass of **Circle**

```

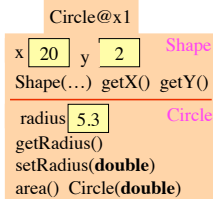
/** An instance represents circle at point in plane */
public class Circle extends Shape {
    all declarations as before except
    /** Constructor: new Circle of radius r at (x, y) */
    public Circle(double r, x, y) {
        radius = r;
    }
}
    
```

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()`

29



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() {}`

`/** 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 */`

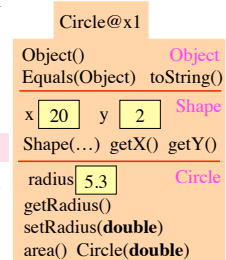
```

public boolean equals(Object ob)
    
```

`c.equals(d)` is **true**

`c.equals(new Circle(...))` is **false**

Page C-18



30

Example of overriding: toString

Override an inherited method: define it in subclass

Put in class Shape

```

/** return representation of this */
public @Override String toString() {
    return "(" + x + ", " + y + ")";
}
c.toString() calls overriding method,
one nearest to bottom of object
c.toString() is "(20, 2)"

```

Do not override a field! Useless. Called shadowing. Not used in 2110

Don't need @Override. Helps catch errors. Use it.

Circle@x1

```

Object()      Object
Equals(Object) toString()
x 20 y 2     Shape
toString()
Shape(...) getX() getY()
radius 5.3   Circle
getRadius()
setRadius(double)
area() Circle(double)

```

c Circle@x1

Page C-12 31

toString() is special in Java

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

Put in class Shape

```

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

```

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

```

System.out.println("c is: " + c);
prints
"c is (20, 2)"

```

Circle@x1

```

Object()      Object
Equals(Object) toString()
x 20 y 2     Shape
toString()
Shape(...) getX() getY()
radius 5.3   Circle
getRadius()
setRadius(double)
area() Circle(double)

```

c Circle@x1

Page B-17 32

Calling overridden method

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

Put in class Circle

```

/** return representation of this */
public @Override String toString() {
    return "Circle radius " +
        radius + " at " +
        super.toString();
}

```

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

Circle@x1

```

Object()      Object
Equals(Object) toString()
x 20 y 2     Shape
toString()
Shape(...) getX() getY()
radius 5.3   Circle
getRadius()
setRadius(double)
area() Circle(double)

```

c Circle@x1

Page C-12 33

Casting among class-types

```

(int) (5.0 / 3) // cast value of expression from double to int
(Shape) c // cast value in c from Circle to Shape

```

Explain, using this situation

```

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

```

Type of variable

Object

Shape

Circle

Circle@x1

```

Object()      Object
Equals(Object) toString()
x 20 y 2     Shape
toString()
Shape(...) getX() getY()
radius 5.3   Circle
getRadius()
setRadius(double)
area() Circle(double)

```

Page C-23, but not good 34

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

Circle@x1

```

... Object wider
... Shape
... Circle narrower

```

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

e Circle@x1 Object

d Circle@x1 Shape

c Circle@x1 Circle

(Object) c widening cast, may be done automatically

(Circle) e narrowing cast, must be done explicitly

Page C-23, but not good 35

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

Circle@x1

```

Object()      Object
Equals(Object) toString()
x 20 y 2     Shape
toString()
Shape(...) getX() getY()
radius 5.3   Circle
getRadius()
setRadius(double)
area() Circle(double)

```

e Circle@x1 Object

d Circle@x1 Shape

Page C-23, not good 36

More on the perspective

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!!!

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 (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)

Spec says return false if ob not a Shape.
 That's what if-statement does

```
/** 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
 c instanceof C true iff object
 c has a partition for class C

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 == ob.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[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();
    ...
}
```

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

```
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`

Parameter: declared in () of method header. Created during call before exec. of method body, discarded when call completed. Initial value is value of corresp. arg of call. Scope: body.

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, MAX_VALUE, toString(int), toBinary(int), valueOf(String), parseInt(String)

Why "wrapper" class?

sandwich wrapper wiggly wrapper `int` wrapper

A wrapper wraps something

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
<code>int</code>	Integer
<code>long</code>	Long
<code>float</code>	Float
<code>double</code>	Double
<code>char</code>	Character
<code>boolean</code>	Boolean

Wrapper class has:

- Instance methods, e.g. equals, constructors, toString,
- Useful static constants and methods.

```
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 `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;`

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

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

	<code>[]@x3</code>
0	5
1	7
2	4
3	-2

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.

<code>a0</code>
length 4
0 5
1 7
2 4
3 -2

`x [a0] int[]`

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;`

`x [null] int[]`

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

`x [a0] int[]`

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

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

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

Arrays

<code>a0</code>
0 0
1 0
2 0
3 0

<code>a0</code>
0 -4
1 0
2 5
3 0

<code>a0</code>
0 -4
1 0
2 6
3 -8

Array initializers

Instead of

`int[] c= new int[5];`
`c[0]= 5; c[1]= 4; c[2]= 7; c[3]= 6; c[4]= 5;`

Use an array initializer:

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

No expression between brackets `[]`.

array initializer: 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

<code>a0</code>
5
4
7
6
5

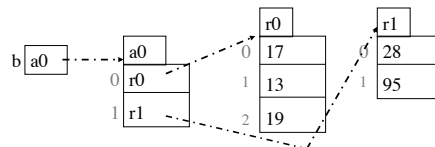
Ragged arrays: rows have different lengths

`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]`.



```

/** = 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
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

```

55

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();
b.set(new Integer(35));
Object x= b.get();
... (Integer) x ...

```

```

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

```

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.

56

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;
    }
    ...
}

```

Replace type Object everywhere by T

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

57

Can extend only one class

```

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

```

if we allowed multiple inheritance, which m used?

```

public class C1 {
    public int m() {
        return 2;
    }
    ...
}
public class C2 {
    public int m() {
        return 3;
    }
    ...
}

```

58

Can extend only one class

```

public class C extends C1, 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.

59

Interface declaration and use of an interface

```

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

```

C must override all methods in C1 and C2

```

public interface C1 {
    int m();
    int p();
    int FF= 32;
}
public interface C2 {
    int m();
    int q();
}

```

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 { ... }

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

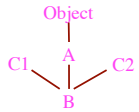
60

Casting with interfaces

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

b = new B();
What does object b look like?

Draw b like this, showing only names of partitions:



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

Add C1, C2 as new dimensions:

61

Look at: interface java.lang.Comparable

```
/** 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
...

When a class implements Comparable it decides what < and > mean!

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

6

62

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

Note TimeOfDay used here

Note: Class implements Comparable

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.

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;
    }
}
```

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

Note use of function compareTo

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

64

Exceptions

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

Happened in C.main on line 7

The "Exception" that is "thrown"

65

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

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

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)
at java.lang.Integer.parseInt(Integer.java:499)
at C.main(C.java:6)
```

called from C.main, line 6

called from line 499

called from line 458

Found error on line 48

3.2 not an int

See stack of calls that are not completed!

66

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**.

67

Exceptions and Errors

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

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

Exception

RuntimeException

ArithmeticException

Error

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

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

Do nothing with these

You can "handle" these

68

Creating and throwing and Exception

Class: `public class Ex {`

04 `public static void main(...) {`

05 `second();`

06 `}`

07 `public static void second()`

08 `third();`

09 `}`

10 `public static void third()`

11 `int x = 5 / 0;`

12 `}`

13 `}`

14 `}`

15 `}`

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)
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)
          
```

69

Throw statement

Class: `public class Ex {`

04 `public static void main(...) {`

05 `second();`

06 `throw new`

07 `ArithmeticException("/ by zero");`

08 `third();`

09 `}`

10 `public static void second()`

11 `third();`

12 `int x = 5 / 0;`

13 `throw new`

14 `ArithmeticException("I threw it");`

15 `}`

16 `}`

17 `}`

Call

Same thing, but with an explicit throw statement

Output

```

ArithmeticException: / by zero
at Ex.third(Ex.java:9)
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)
          
```

70

How to write an exception class

```

/** 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();
    }
}
  
```

71

The "throws" clause

```

/** 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("mine");
    }
}
  
```

Throw Exception that is not subclass of RuntimeException? May need throws clause

If Java asks for a throws clause, insert it. Otherwise, don't be concerned with it.

72

Try statement: catching a thrown exception

```

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

```

try-block

catch-block

class-name that is a subclass of Throwable

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** 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.

73

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**

74

Junit testing class looks like this:

```

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**

75

What to put in a test method

```

...
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));
    }
}

```

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

assertEquals(expected value, computed value);

76

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.

77

Testing a constructor

```

...
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());
    }
}

```

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.

Note: purpose of procedure is to test constructor, but the method also tests the getter methods.

This code creates a new objects and tests whether *all* fields are properly set.

78

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.

79

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 is first created, whatever).

Just use local variables where needed in a testing class.

80

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}
```

could be private, or any access modifier

new keyword

The constants of the class are Clubs, Diamonds, Hearts, Spades

81

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:
    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

82

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

83

Miscellaneous points about enums

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

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);
```

You can see that `toString` in object `Clubs` returns the string "Clubs"

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`

84

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

<code>Suit.Clubs.ordinal()</code>	is	0
<code>Suit.Diamonds.ordinal()</code>	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.

85