6.092 - Introduction to Programming in Java

Lecture 1: Types, Variables, and Operators

Program Structure

```
class CLASSNAME {
  public static void main(String[] arguments) {
    STATEMENTS
  }
}
```

Output

```
System.out.println("output");
```

Types

- boolean: **true, false**
- int: 0,-1, 46
- double: 3.145, -1.5
- String: “hello”, “world”

Variables

```
TYPE NAME;

int a;
String foo;
```

Assignment

Using = to give variables a value.

```
foo = "hello";
a = 10;
int b = 10;
```

Operators

Operators → **+ - * /**

Priority

1. Parentheses
2. Multiplication and Division
3. Addition and Subtraction

String Concatenation

Concatenation → string + string

```
String text = "hello " + "world";
text += " !";
```

Lecture 2: More Types, Methods, Conditionals

Conversion

Division operates differently on integers and on doubles!

5/2 = 2

5/2.0 = 2.5 int → double (implicit)

Java supports implicit and explicit conversion.

```
double d = 5/2.0;
int a = (int) 5/2.7;
```

Methods

```
//Declaration
public static void NAME() {
  STATEMENTS
}

//To call a method:
NAME();
```

Parameters

```
//Declaration
public static void NAME(TYPE NAME) {
  STATEMENTS
}

//To call:
NAME(EXPRESSION);
```

Multiple Parameters

```
//Declaration
public static void NAME(TYPE NAME, TYPE NAME) {
  STATEMENTS
}

//To call:
NAM(ARG1,ARG2);
```

Return Values

```
public static TYPE NAME() {
  STATEMENTS
  return EXPRESSION;
}
```

void means “no type”

Variable Scope

Variables live in the block ({ ... }) where they are defined (scope).

Method parameters are like defining a **new** variable in the methods.

Mathematical Functions

```
Math.sin(x)
Math.cos(Math.PI / 2)
Math.pow(2,3)
Math.log(Math.log(x+y))
```

Conditionals

Conditional Operators: >, <, >=, <=, ==

Booleans Operators: && (logical AND), || (logical OR)

if statement

```
if (CONDITION) {
  STATEMENTS
}
```

else statement

```
if (CONDITION) {
  STATEMENTS
} else {
  STATEMENTS
}
```

else if statement

```
if (CONDITION) {
  STATEMENTS
} else if (CONDITION) {
  STATEMENTS
} else if (CONDITION) {
  STATEMENTS
} else {
STATEMENTS
}
```

Conversion by method

int to String:

```
String five = 5; // ERROR!
String five = Integer.toString (5);
Stringfive=“”+5; //five=“5”
```

String to int:

```
int foo = “18”; // ERROR!
int foo = Integer.parseInt (“18”);
```

Note:

Do not call == on doubles! EVER

Example

double a = Math.cos(Math.PI / 2);

double b = 0.0;

a == b → false

a = 6.12 * 10^(-17) (very small) ≠ 0

Lecture 3: Loops and Arrays

Good Programming Style

1. Use good (meaningful) names.
2. Use indentation.
3. Use whitespaces and blank lines.
4. Do not duplicate tests → if doing the same thing.

Loops

Loop operators allow to loop through a block of code.

while operator

```
while (CONDITION) {
  STATEMENTS
}

int i = 0;
while (i < 3) {
  System.out.println(“Rule #“ + i);
  i = i+1;
}
```

for operator

```
for (INITIALISATION;CONDITION;UPDATE) {
  STATEMENTS
}

for (int i = 0; i < 3; i++) {
 System.out.println(“Rule #“ + i);
}
```

break: terminates a for or while loop.

continue: skips the current iteration of the loop and proceeds directly to the next iteration.

Nesting or Embedding of loops possible.

```
for (int i = 0; i < 3; i++) {
  for (int j = 2; j < 4; j++) {
    System.out.println (i + “ “ + j);
  }
}
```

Scope of the variable defined in the initialisation: respective for block.

Arrays

An array is an indexed list of values.

All elements of an array must have the same type.

Declaration

defined using TYPE[ ]

```
int[] values; //array of int
int[][] values; //int[] is the type
```

to create an array of given size, use **new.**

```
int[] values = new int[5];
```

Array Initialisation

Use { ... }

can ONLY be used when you declare the variable.

```
int[] values = {12,24,36};
```

Accessing Arrays

```
values[index] // index->[0,len-1]
```

Array Length

```
values.length //return the size of the array
```

Looping through an array

```
int[] values = new int[5];
for (int i=0; i<values.length; i++) {
  values[i] = i;
  int y = values[i] * values[i];
  System.out.println(y);
}
```

Lecture 4: Objects and Classes

Classes

```
public class Baby {
  FIELDS

  METHODS
}

public class Baby {
  String name;
```

```
  boolean isMale;
  double weight;
  double decibels;
  int numPoops = 0;

  void poop() {
    numPoops += 1;
    System.out.println(“Dear mother, ”+ “I have pooped. Ready the diaper.”);
  }
}
```

- Class names are Capitalised.
- 1 class = 1 file
- Having a main method means class can be run.

Class Instance/Object

```
Baby myBaby = new Baby(); //class instance
```

Constructors

Constructor name == Class name

Nor return type.

Usually initialise fields.

Need at least one constructor, default given below

```
CLASSNAME () {} //default constructor

public class CLASSNAME{
  CLASSNAME ( ) {
  }
  CLASSNAME ([ARGUMENTS]) {
  }
}

CLASSNAME obj1 = new CLASSNAME();
CLASSNAME obj2 = new CLASSNAME([ARGUMENTS])
```

Using Classes

Accessing Fields

```
object.FIELDNAME

Baby shiloh = new Baby(“Shiloh Jolie-Pitt”, true);
System.out.println(shiloh.name);
System.out.println(shiloh.numPoops);
```

Calling Methods

```
object.METHODNAME([ARGUMENTS])

Baby shiloh = new Baby(“Shiloh Jolie-Pitt”, true);
shiloh.sayHi();
shiloh.eat(1);
```

References vs Values

Primitive types are basic Java types

Actual value stored in variable.

Reference type are arrays and objects.

How does Java store objects?

- Objects are too big to fit in a variable.

  Stored somewhere else.

  Variables stores a number that locates the object.
- Object's location is called a reference.
- Using = updates the reference.

  baby1 = baby2

static

- Applies to fields and methods.
- Means the field/method
  - is defined for the class declaration.
  - is not unique for each instance.
- non-static methods can reference static methods but not the other way around.
- common for all objects.

```
static int numBabiesMade = 0; //static field

static void cry() {
  System.out.println(name);
}
```

Lecture 5: Access Control, Class Scope, Packages, Java API

Access Control

Public vs. Private

- **public:** others can use this
- **private:** only the class can use this

public/private applies to any field or method.

```
public class CreditCard {
  private string cardNumber;
  private double expenses;
  public void charge(double amount) {
    expenses += amount;
  }
}
```

Why Access Control?

1. Protect private information.
2. Clarify how others should use your class.
3. Keep implementation separate from interface.

Class Scope

Just like methods, variables are accessible inside { ... }

method-level scope:

```
void method(int arg1) {
  int arg2 = arg1 + 1;
}
```

class-level scope:

```
class Example {
int memberVariable;
void setVariable(int newVal) {
  memberVariable += newVal;
}
}
```

‘this’ keyword

Clarifies scope.

Means ‘my object’

```
this.memberVariable
```

No confusion between variable defined in method and object variable in class.

Packages

Each class belongs to a package.

Classes in the same package serve a similar purpose.

Packages are just directories.

Classes in other packages need to be imported.

Defining Packages

```
package path.to.package.foo;

class Foo {
 ...
}
```

Using Packages

```
import path.to.package.foo.Foo;
import path.to.package.foo.*;
```

Why Packages?

Combine similar functionality.

Ex. libraries.Library and libraries.Book

Separate similar names.

Ex. shopping.List and packing.List

Special Packages

All classes “see” classes in the same package (no import needed).

All classes “see” classes in java.lang.

Ex. java.lang.String, java.lang.System

Java API

Java includes lots of packages/classes.

Reuse classes to avoid extra work.

> Java Platform SE 6
>
> http://java.sun.com/javase/6/docs/api/

ArrayList

Modifiable list.

Internally implemented with arrays.

Features

- Get/put items by index
- Add items
- Delete items
- Loop over all items

```
import java.util.ArrayList //need to import this to use ArrayList

ArrayList<Book> books = new ArrayList<Book>();
```

```
books.add(b); //insert new book
books.size(); //length of array
books.get(i); //get element in index 'i'
books.set(0,b); //replace 0th index element with b
books.remove(1); //remove elemetn at index 1
```

Sets

Like an ArrayList, but

- Only one copy of each object.
- No array index.

Features

- Add objects to the set.
- Remove objects from the set.
- Is an object in the set?

TreeSet: sorted (low to high)

HashSet: unordered (pseudo-random)

```
import java.util.TreeSet;

TreeSet<String> strings = new TreeSet<String>();
```

```
strings.add("Evan");
strings.remove("Eugene");
```

Maps

Stores (key, value) pair of objects.

Look up the key, get back the value

TreeMap: sorted (low to high)

HashMap: unordered (pseudo-random)

```
import java.util.HashMap;

HashMap<String, String> strings = new HashMap<String, String>()
```

```
strings.put("Adam", "email3@mit.edu”);
strings.size();
strings.get("Eugene");
strings.remove("Evan");
strings.keySet();
strings.values();
```

Looping

```
for (String s : strings) { //for ArrayList and Sets
  System.out.println(s);
}

for (Map.Entry<String, String> pairs : strings.entrySet()) { //for Maps
  System.out.println(pairs);
}
```

Warning!

Using TreeSet/TreeMap?

> Read about comparable interface.

Using HashSet/HashMap?

> Read about equals, hashCode methods.

Note: This only matters for classes you build, not for java built-in types.

Lecture 6: Design, Debugging, Interfaces

Good Program Design

A good program has:

- no errors
- easy to understand
- easy to modify/extend
- good performance

For consistent code use the **style guidelines** mentioned below.

Naming

Variables: Nouns, lowercase first letter, capitals separating words
x, shape, highScore, fileName

Methods: Verbs, lowercase first letter
getSize(), draw(), drawWithColor()

Classes: Nouns, uppercase first letter
Shape, WebPage, EmailAddress

Good Class Design

Good classes: easy to understand and use

- Make fields and methods private by default
- Only make methods public if you need to
- If you need access to a field, create a method:

```
public int getBar() { return bar; }
```

Debugging

The process of finding and correcting an error in a program.

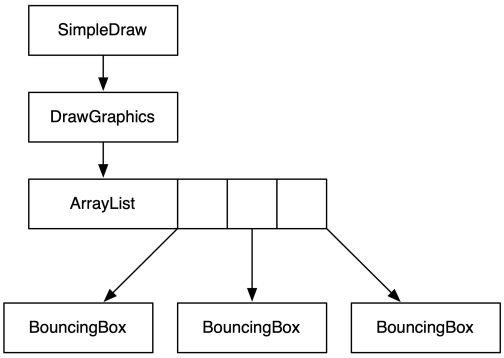
Step 1: Don't make mistakes

Reuse: find existing working code

Design: think before you code

Pseudocode: A high-level, understandable description of what a program is supposed to do.

Use a visual design for objects, or how a program works.



Best Practices: recommended procedures/techniques to avoid common problems

Step 2: Find mistakes early

Easier to fix errors the earlier you find them.

- Test design, implementation
- Detect potential error
- Check you work: assertions

Interval Testing (a,b)

Check all possible boundary cases.

What if a > b or a == b?

Eclipse Warnings

May not be a mistake, but it likely is.

Always fix all warnings.

Assertions

Verify that code does what you expect.

if true: nothing happens.

if false: program crashes with error.

This is disabled by default (enabled with -ea)

```
assert difference >= 0;
```

Step 3: Reproduce the Error

- Figure out how to repeat the error
- Create a minimal test case

Step 4: Generate Hypothesis

What is going wrong?

What might be causing the error?

Step 5: Collect Information

If x is the program, how can you verify?

System.out.println( ) is very powerful.

Eclipse debugger can help.

Step 6: Examine Data

Examine your data.

Is the hypothesis correct?

Why use Methods?

Write and test code one, use it multiple times: **avoid duplication**.

Use it without understanding how it works: **encapsulation/information hiding.**

Why use Objects?

Combine a related set of variables and methods.

Java Interfaces

Manipulate objects, without knowing how they work.

Useful when you have similar but not identical objects.

Useful when you want to use code written by others.

Defining Interfaces

Set of classes that share methods.

Declare an interface with the common methods.

Can use the interface, without knowing an object's specific type.

```
import java.awt.Graphics;

interface Drawable {
  void draw(Graphics surface);
  void setColor(Color color);
}
```

Implementing Interfaces

Implementation provide complete methods.

```
import java.awt.Graphics;
class Flower implements Drawable {
  // ... other stuff ...
  public void draw(Graphics surface) {
    // ... code to draw a flower here ...
  }
}
```

Notes

Interface mostly has only methods.

Do not provide code, only the definition.

A class can implement any number of interfaces.

Using Interfaces

Can only access stuff in the interface.

```
Drawable d = new BouncingBox(...);
d.setMovementVector(1, 1); //error
//undefined for type Drawable
```

Casting

If you know that a variable holds a specific type, you can use a cast.

```
Drawable d = new BouncingBox(...);
BouncingBox box = (BouncingBox) d;
box.setMovementVector(1, 1);
```

Remember

You must implement **all** the methods.

All fields are **final** (cannot be changed).

```
public interface ICar {
  boolean isCar = true;
  int getNumWheels();
}
```

Lecture 7: Inheritance, Exceptions, File I/O

Inheritance

```
public class Dude {
  public String name;
  public int hp = 100
  public int mp = 0;
  public void sayName() {
    System.out.println(name);
  }
  public void punchFace(Dude target) {
    target.hp -= 10;
  }
}
```

Now, creating a **Wizard**

A Wizard does and has every thing a Dude does and more!

For creating a wizard class you don't have to copy & paste.

```
public class Wizard extends Dude {
}
```

- Wizard can use everything* the Dude has! **(fields)**

  ```
  wizard1.hp += 1;
  ```

- Wizard can do everything Dude can do! **(methods)**

  ```
  wizard1.punchFace(dude1);
  ```

- You can use Wizard like a Dude too!

  ```
  dude1.punchface(wizard1);
  ```

- Can augment a Wizard.

  ```
  public class Wizard extends Dude {
    ArrayList<Spell> spells;

    public class cast(String spell) {
      // cool stuff here
      ...
      mp -= 10;
    }
  }
  ```

***except for **private** fields and methods.

Inheriting from Inherited Classes

```
public class GrandWizard extends Wizard {
  public void sayName() {
    System.out.println("Grand Wizard" + name);
  }
}

grandWizard1.name = "Flash";
grandWizard1.punchFace(dude1); // A
((Dude) grandWizard1).sayname(); // B
```

a. What Java does when it sees A

   Search for the methods/fields starting from the bottom to the top of the hierarchy of ancestors.

   GrandWizard → Wizard → Dude

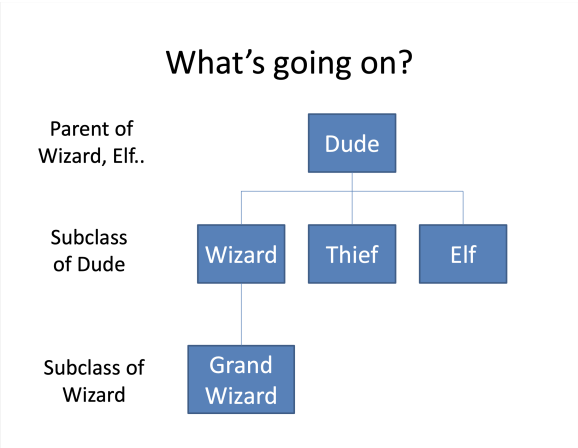
b. What Java does when it sees B

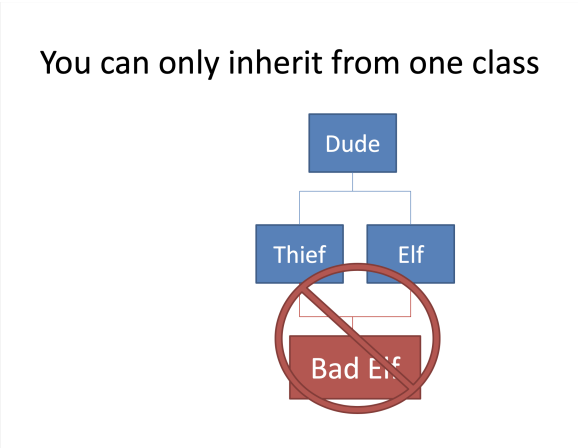
   a. Cast to Dude tells Java to start looking in Dude

   b. Look for sayName( ) in Dude class

   c. Found it! Call sayName( )

Inheritance Structures





You can only inherit from one class.

Inheritance Summary

- Class A extends B { } == A is a subclass of B
- A has all the fields and methods that B has
- A can add it's own fields and methods
- A can have only 1 parent
- A can replace a parent's method by re-implementing it
- If A doesn't implement something Java searches ancestors

Exceptions

Event that occurs when something "unexpected" happens

Example:

- null.someMethod()
- (new int[1])[1] = 0;
- int i = "string";

Types

NullPointerException

ArrayIndexOutOfBoundsException

ClassCastException

RuntimeException

Why use an Exception?

To tell the code using your method that something went wrong.

Helps in debugging and understanding control flow.

How do exceptions "happen"?

Java doesn't know what to do, so it

1. Creates an Exception object
2. Includes some useful information
3. "throws" the Exception

Exception is a class, you can inherit from it!

```
public clazss MyException extends Exception {
  ...
}
```

Warn Java about the Exception

```
public Object get(int index) throws ArrayOutOfBoundsException {
  if (index < 0 || index >= size()) {
    throw new ArrayOutOfBoundsException(“”+index);
  }
}
```

- **throws** tells Java that *get* function may throw the exception.
- **throw** actually throws the Exception

Java now excepts code that calls *get* to deal with the exception by

1. Catching it
2. Rethrowing it

Catching It

try to run some code that may throw an exception

Tell Java what to do if it sees the exception (**catch**).

```
try {
  get(-1);
} catch (ArrayOutOfBoundsException err) {
  System.out.println(“oh dear!”);
}
```

Rethrowing It

Maybe you don’t want to deal with the Exception.

Tell Java that your method throws it too.

```
void doBad() throws ArrayOutOfBoundsException {
  get(-1);
}
```

What if no one catches the Exception?

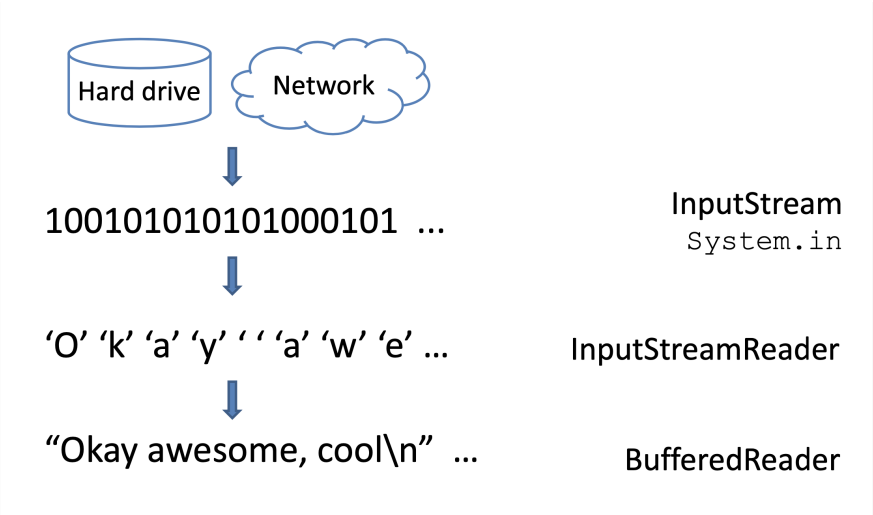
Java will print an **error** message.

I/O

Output

```
System.out.println("some string");
```

The Full Picture



InputStream

- InputStream is a stream of bytes
  - Read one byte after another using read()
- A byte is just a number
  - Data on you hard drive is stored in bytes.
  - Bytes can be interpreted as characters, numbers..

```
InputStream stream = System.in;
```

InputStreamReader

- Reader is a class for character streams.
  - Read one character after another using read()
- InputStreamReader takes an InputStream and converts bytes to characters
- Still inconvenient
  - Can only read a character at a time

```
new InputStreamReader(stream)
```

BufferedReader

- BufferedReader buffers a character stream so you can read line by line
  - String readLine( )

```
new BufferedReader(new InputStreamReader(System.in));
```

User Input

```
InputStreamReader ir = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader(ir);

br.readLine();
```

FileReader

- FileReader takes a text file
  - Converts it into a character stream
  - FileReader(”PATH TO FILE”);
- Use this + BufferedReader to read files!

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ReadFile {
  public static void main(String[] args) throws IOException{
    // Path names are relative to project directory (Eclipse Quirk )
    FileReader fr = new FileReader("./src/readme");
    BufferedReader br = new BufferedReader(fr);
    String line = null;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
    br.close();
  }
}
```