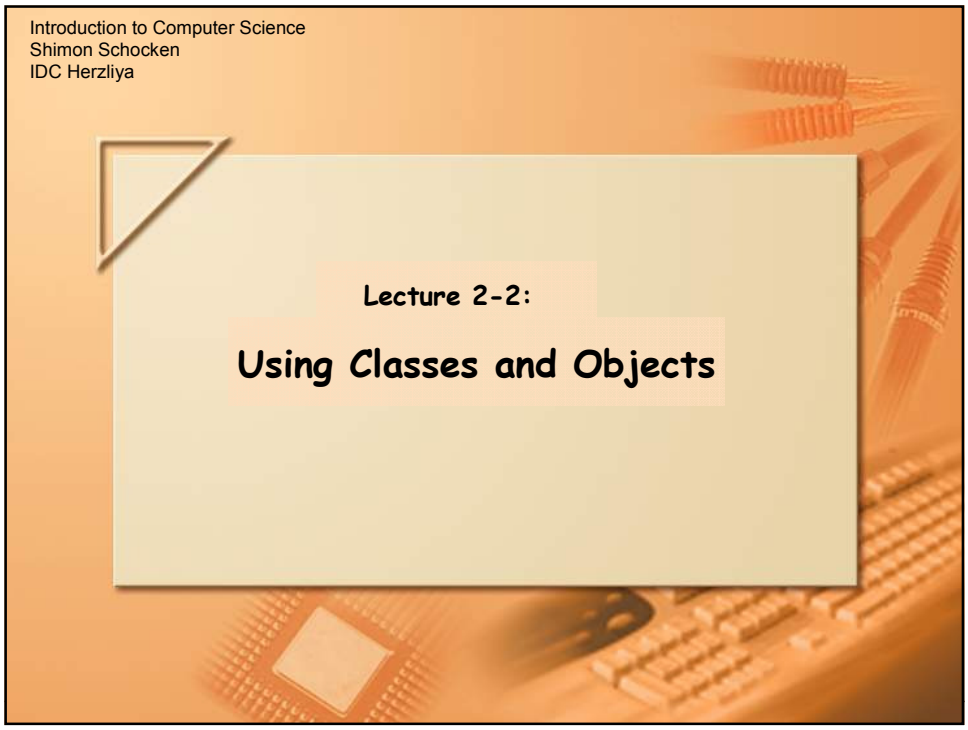


Classes

When you write a Java program, you often have to use the services of other classes; classes that you wrote, and classes written by other people.

A class is a self-contained module of code. Each class code is stored in a separate file. If the class is called "`Car`", then the Java code is stored in `Car.java` and the compiled bytecode is stored in `Car.class`

Classes provide functionality: they enable us to carry out all sorts of computations, and to create and manipulate *objects*

This functionality has many variants. For example, we can think of classes like `Car`, `CarRace`, `CarHistory`, and so on. These classes are designed to represent and do very different things.

Two viewpoints on classes:

- Client view: how to use existing classes — This lecture
- Server view: how to design, implement, and maintain classes — Later lectures

Outline

- Static classes
  - `Math`
  - `Arrays` (later)
- Classes that represent types (examples)
  - `Fraction`
  - `bankAccount`
  - `Turtle`
  - `String`
- Classes that generate things (examples)
  - `Random`

(More kinds of classes – later in the course)

- Packages

Math class

Represented in Java as:

π	`Math.PI`
$\sqrt{x}$	`Math.sqrt(x)`
$\log_{10} x$	`Math.log(x)`
$Min(x, y)$	`Math.min(x,y)`
$\lvert x \rvert$	`Math.abs(x)`
x^y	`Math.pow(x,y)`
	`...`

Static variables and static methods in the static `Math` class

Syntax for using a static variable:

```
ClassName.variableName
```

Syntax for invoking a static method:

```
ClassName.methodName(parameters)
```

Example:

```
// Variable r holds the value of some radius
System.out.println( "Circle area = " + Math.PI * Math.pow(r,2) );
```

How do you know which variables and methods the `Math` class offers and how to use them?

You consult the `Math` class interface.

Math class interface (partial)

Field Summary	
static double	E The double value that is closer than any other to *e*, the base of the natural logarithms.
static double	PI The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Summary	
static double	sin(double a) Returns the trigonometric sine of an angle.
static double	sinh(double x) Returns the hyperbolic sine of a double value.
static double	sqrt(double a) Returns the correctly rounded positive square root of a double value.
static double	tan(double a) Returns the trigonometric tangent of an angle.
static double	tanh(double x) Returns the hyperbolic tangent of a double value.
static double	toDegrees(double angrad) Converts an angle measured in radians to an approximately equivalent angle measured in degrees.
static double	toRadians(double angdeg) Converts an angle measured in degrees to an approximately equivalent angle measured in radians.

A class interface gives all the information you needs in order to use the class services

In OOP jargon, "static" means "not associated with any object"

Static classes

Static classes are sometimes called "utility classes"

A utility class provides a library of methods that have something in common

Examples:

- `Math`: a library of mathematical operations
- `Arrays`: a library of array-oriented operations

(Disclaimer: The rest of this slide will make sense only at the end of this lecture)

In OOP, static classes are the exception: they go against the "OOP spirit", since they involve no objects

In a pure OOP world, everything is an object. So:

- Instead of saying `z = Math.sqrt(x)` an OOP purist would say `z = x.sqrt()`
- Instead of saying `z = x + y` an OOP purist would say `z = x.add(y)`
- Etc. – we'll get back to this comment later.

Outline

- Static classes
  - `Math`
  - `Arrays` (later)
- Classes that represent types (examples)
  - `Fraction`
  - `bankAccount`
  - `Turtle`
  - `String`
- Classes that generate things (examples)
  - `Random`

(More kinds of classes – later in the course)

- Packages

Classes that represent type abstractions

Primitive types:

Defined by the Java language:
- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`
- `boolean`
- `char`

Abstract data types (ADT)

Java library classes:
- `BigDecimal`
- `Point`
- `String`
- `Date`
- `InetAddress`
- `Set`
- `...`

User defined classes:
- `Fraction`
- `Complex`
- `Matrix`
- `Point`
- `Graph`
- `Polynomial`
- `...`

Abstractions and classes:

The world around us consists of much more than Java's 8 primitive types

To capture this richness, OOP languages are made to be extensible:
new data types are defined as needed, by software architects

In OOP, type abstractions are represented using:

- Classes ← This lecture
- Interfaces ← Later lectures

Fractions

Requirements analysis (first approximation)

Needed: some mechanism for representing and manipulating fractions.
For example, given 1/2 + 1/3, we wish to compute and return 5/6.

A fraction can be characterized by two integers: numerator and denominator.

Things we want to do with fractions:

- Construction (how should we handle a zero denominator?)
- Addition
- Multiplication
- Division
- Displaying
- Check if two fractions are equal
  (is 1/2 the same as 3/6?)
- Etc.

Design decision:
We will always represent fractions in their reduced version.

Fraction abstraction

Fraction class API

Constructor Summary

`Fraction(int numerator, int denominator)`	Constructs a Fraction which is the reduced division of the two parameters.

Method Summary

`Fraction`	`add(Fraction other)` Returns a Fraction which is the sum of this Fraction and the other one.
`Fraction`	`divide(Fraction other)` Returns a Fraction which is the division of this Fraction and the other one.
`boolean`	`equals(java.lang.Object obj)` Determines whether or not this Fraction is equal to the other one.
`int`	`getDenominator()` Returns the denominator of this Fraction as an int.
`int`	`getNumerator()` Returns the numerator of this Fraction as an int.
`int`	`hashCode()` Returns a hash code for this Fraction.
`Fraction`	`multiply(Fraction other)` Returns a Fraction which is the product of this fraction and the other one.
`Fraction`	`reciprocal()` Returns a Fraction which is the reciprocal of this Fraction.
`java.lang.String`	`toString()` Returns a string representation of this Fraction in the form "numerator/denominator"

Fraction abstraction

Fraction class API

Constructor Summary

`Fraction(int numerator, int denominator)`
Constructs a Fraction which is the reduced division of the two parameters.

Method Summary

Fraction	add(Fraction other) Returns a Fraction
Fraction	divide(Fraction othe Returns a Fraction
boolean	equals(java.lang.Obj Determines whethe
int	getDenominator() Returns the denomi
int	getNumerator() Returns the numera
int	hashCode() Returns a hash cod
Fraction	multiply(Fraction ot Returns a Fraction
Fraction	reciprocal() Returns a Fraction
java.lang.String	toString() Returns a string representation of this Fraction in the form "numerator/denominator"

API = functional abstraction

Architect's view:

Express and structure the desired functionality in a way that other people will find easy and natural to use

Client view:

- How do I use this class in my programs?
- I don't care how the class is implemented

Very important design principle:

Keep abstraction and implementation separate.

Using Fractions

server

Constructor Summary

`Fraction(int numerator, int denominator)`
Constructs a Fraction which is the reduced division of the two parameters.

Method Summary

Fraction	add(Fraction other) Returns a Fraction which is the sum of this Fraction and the other one.
Fraction	divide(Fract Returns a
boolean	equals(java. Determin
int	getDenominat Returns t
int	getNumerator Returns t
int	hashCode() Returns a
Fraction	multiply(Fra Returns a
Fraction	reciprocal() Returns a
java.lang.String	toString() Returns a string representation of this Fraction in the form "numerator/denominator"

client code

```
public class FractionDemo {
  public static void main (String args[]) {
    Fraction a = new Fraction(1,2);
    Fraction b = new Fraction(2,3);
    System.out.println("a = " + a.toString());
    System.out.println("b = " + b);  // same as b.toString()
    System.out.println("a + b = " + a.add(b));
    System.out.println("a * b = " + a.multiply(b));
    if (a.add(b).equals(b.add(a)))
       System.out.println("addition seems to be commutative");
  }
}
```

Output

```
a = 1/2
b = 2/3
a + b = 7/6
a * b = 1/3
addition seems to be commutative
```

Creating objects

server

Constructor Summary

`Fraction(int numerator, int denominator)`
Constructs a Fraction which is the reduced division of the two parameters.

client code

```
...
Fraction a = new Fraction(1,2);
Fraction b = new Fraction(2,3);
...
int c = 5;
...
```

Object declaration syntax:

```
ClassName varName  = new  ClassName(parameters);
```

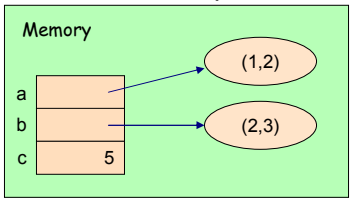
The parameter values serve to initialize the object's state.

A constructor is a method:

It takes 0 or more parameters and returns a value

This value is the base address of the newly created object

Behind the scene view of object variables:



a and b are called "object variables" or "reference variables"

c is called "primitive variable"

Primitive variables store values

Object variables store base addresses of objects.

Outline

- Static classes
  - `Math`
  - `Arrays` (later)
- Classes that represent types (examples)
  - `Fraction`
  - `bankAccount`
  - `Turtle`
  - `String`
- Classes that generate things (examples)
  - `Random`

(More kinds of classes – later in the course)

- Packages

Bank account abstraction

Requirements analysis (first approximation)

A bank account is characterized by:
owner, balance, and a unique identifying number.

The account number is assigned automatically when the account is opened.

Things that we want to do with a bank account:

- Show its current balance
- Deposit money
- Withdraw money
- Transfer money to another account.
- Some more operations, to be defined later.

BankAccount abstraction

Constructor Summary

BankAccount(java.lang.String owner, double balance)	Constructs a new BankAccount with a specified owner and balance and allocates to it a unique bank account number.

Method Summary

void	deposit(double amount) Handles a deposit.
boolean	equals(java.lang.Object other) Determines weather this BankAccount equals the other BankAccount.
double	getBalance() Returns the balance of this bankAccount.
int	getNumber() Returns the number of this bankAccount.
java.lang.String	getOwner() Returns the owner of this bankAccount.
int	hashCode() Returns a hash code for this BankAccount.
java.lang.String	toString() Returns a textual description of this BankAccount
void	transferTo(double amount, BankAccount other) Handles a transfer of money from this bank account to the other bank account.
void	withdraw(double amount) Handles a withdrawl.

BankAccount is not unlike a data type: it describes data and operations.

Creating and manipulating objects

BankAccount class interface: (same)

server

Constructor Summary

`BankAccount(java.lang.String owner, double balance)`
Constructs a new BankAccount with a specified owner and balance and allocates to it a unique bank account number.

Method Summary

void	`deposit(double amount)` Handles a deposit.
boolean	`equals(java.lang.Obj` Determines weather
double	`getBalance()` Returns the balance
int	`getNumber()` Returns the number
java.lang.String	`getOwner()` Returns the owner
int	`hashCode()` Returns a hash cod
java.lang.String	`toString()` Returns a textual de
void	`transferTo(double an` Handles a transfer
void	`withdraw(double amount)` Handles a withdrawl.

Client code

```
public class BankAccountDemo {
   public static void main (String args[]) {
       BankAccount aliceAcc = new BankAccount("Alice", 0);
       BankAccount bobAcc = new BankAccount("Bob", 100);
       // ...
       aliceAcc.deposit(900);
       aliceAcc.transferTo(700, bobAcc);
       System.out.println(aliceAcc);
       System.out.println(bobAcc);
       // ...
   }
}
```

Output:

```
1       Alice   200
2       Bob     800
```

- The `BankAccount` class can be viewed as a template for creating and manipulating bank accounts
- Using it, we can create and manipulate as many `BankAccount` *instances*, or *objects*, as needed by the application.

Turtles

server

Constructor Summary

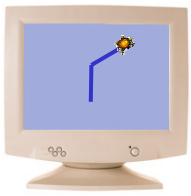
`Turtle()`
Constructs a new turtle.

Method Summary

void	`disableDelay()` Disables the delay of the turtle.
void	`hide()` Hides the turtle.
void	`moveBackward(double units)` Moves the turtle backwards by a given number of
void	`moveForward(double units)` Advances the turtle forwards by a given number of
void	`show()` Shows the turtle.
void	`tailDown()` Lowers the tail of the turtle.
void	`tailUp()` Raises the tail of the turtle.
void	`turnLeft(int degrees)` Turns the turtle counter-clockwise.
void	`turnRight(int degrees)` Turns the turtle clockwise.

client code

```
public class TurtleDrawingDemo {
  public static void main(String[] args) {
    Turtle leonardo = new Turtle();
    leonardo.tailDown();
    leonardo.moveForward(100);
    leonardo.turnRight(60);
    leonardo.moveForward(100);
  }
}
```



Method invocations (aka "method calls")

Code examples (meaningless ...)

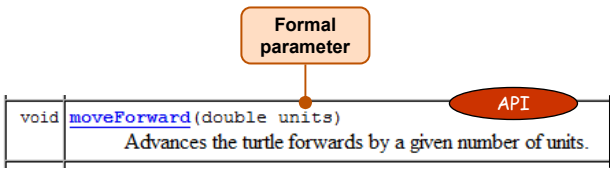
```
leonardo.hide();              // calling a void method with no parameters

leonardo.moveForward(100);    // calling a void method with one parameter

x = Math.sqrt(area);          // calling a static method that returns a value

System.out.println(bobAct.getBalance());     // two method calls

Turtle leonardo = new Turtle();              // calling a constructor

BankAccount bobAct = new BankAccount("Bob", 900)    // calling a constructor
```

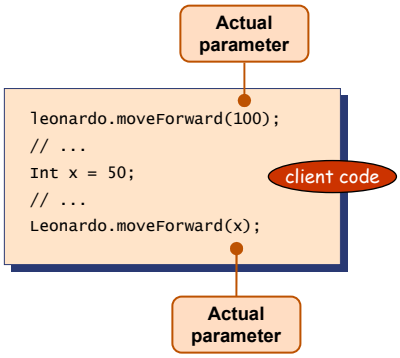
Methods may have parameters, or not

Methods may be invoked on objects, or not

Methods may return values, or not.

Parameters





- **formal parameters**: stated in the method's API
- **actual parameters**: the values passed by the method's caller (aka "arguments").

Two modes of parameter passing

- **Call by value**: the calling code passes the value of the actual parameter
- **Call by reference**: the calling code passes the variable itself, i.e. it's address in memory

Outline

- Static classes
  - `Math`
  - `Arrays` (later)
- Classes that represent types (examples)
  - `Fraction`
  - `bankAccount`
  - `Turtle`
  - `String`
- Classes that generate things (examples)
  - `Random`

(More kinds of classes – later in the course)

- Packages

Strings

Requirements analysis (first approximation)

Needed: A mechanism for representing and manipulating strings of characters like `"IDC"`, `"Los Angeles"`, `"AATTTCCGTG"`, etc.

Things we want to do with strings:

- Concatenation: `city = "Los " + "Angeles"` should yield the string `"Los Angeles"`
- Find where a given string starts in the string: `city.indexOf("ng")` should return 5
- Find what appears in a give location: `city.charAt(5)` should return 'n'
- Etc. – many more similar string manipulation operations

- String processing is prevalent: language parsing, DNA research, web protocols, …
- The Java solution: a built-in `String` class that provides many string processing capabilities.

Strings

String = a sequence of characters, e.g. `"Los Angeles"`

In Java, character strings are represented as `String` objects

```
// Some (meaningless) examples of working with Strings
String name = new String("John Cleese");
String title = "Mr.";   // Special object construction shortcut,
                        //  unique to Strings
String salutation = title + name;   // "Mr. John Cleese"

int year = 2012;
String line = "See you in London in " + year;
line = line + ", if I can afford it.";
System.out.println(line);
// Displays "See you in London in 2012, if I can afford it."
```

- Every string of characters is represented by an object of the `String` class
- `'b'` and `"b"` are not the same thing!

- Strings are concatenated using the + operator
- The + operator is type-sensitive.

```
System.out.println("2 and 3 concatenated: " + 2 + 3);
System.out.println("2 and 3 added: " + (2 + 3));
```

Output:

```
2 and 3 concatenated: 23
2 and 3 added: 5
```


String methods (a small sample out of about 50 of them)

	0	1	2	3	4	5	6	7
City:	T	e	l		A	v	i	v

```
String s, city = "Tel Aviv";
char c;
int n;
c = city.charAt(0);       // 'T'
n = city.length();        // 8
n = city.indexOf('v')     // 5
n = city.indexOf("el")    // 4
s = city.subString(2,4);  // "i A"
s = city.subString(3);    // " Aviv"
s = city.toUpperCase();   // "TEL AVIV"
s = city.replace('v','g');  // "Tel Agig"
```

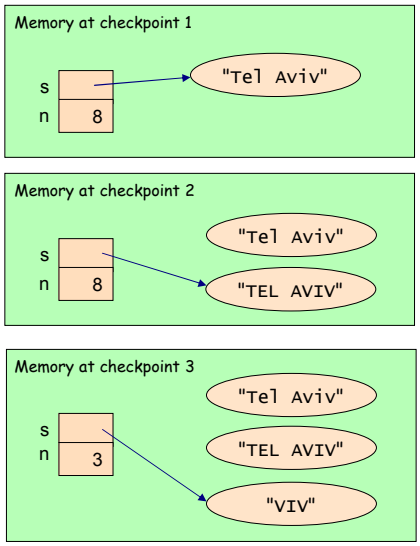
Method overloading: classes often feature several versions of the same method. The difference is in the number and type of the method parameters.

Method overloading: Often used to apply the same or similar computation on different types and numbers of the parameters.

The String class is immutable

String objects are immutable (unchangeable): once a `String` object has been created, neither its value nor its length can be changed

```
String s = "Tel Aviv";
int n = s.length();
// checkpoint 1
s = s.toUpperCase();
// checkpoint 2
s = s.subString(5);
n = s.length();
// checkpoint 3
```



De-referenced objects are reclaimed by a behind-the-scene process called garbage collector

Mutability

Mutable class: a class whose objects can be changed.

Examples: `BankAccount, Turtle`

Immutable class: a class whose objects – once constructed – never change.

Examples: `Fraction, String`

Why some classes are mutable and others are immutable?

The mutability of a class is determined by the class architect when building the class interface; The class user has no choice – she uses what she's got

Architect's best practice:

- Immutable classes are safer and easier to manage
- As a general rule, a good architect strives to minimize access to her objects.

Outline

- Static classes
  - `Math`
  - `Arrays` (later)
- Classes that represent types (examples)
  - `Fraction`
  - `bankAccount`
  - `Turtle`
  - `String`
- Classes that generate things (examples)
  - `Random`

(More kinds of classes – later in the course)

- Packages

Classes that generate things: `Random`

The Java class library includes a class named `Random`, which is part of the package `java.util`

`Random` is used to generate a stream of pseudorandom numbers

More accurately: an instance of the `Random` class is used as a generator designed to generate pseudorandom numbers

Example:

```
// Generates and prints two random numbers
public class RandomNumbersDemo {
   public static void main (String[] args){
      Random rndGenerator = new java.util.Random();
      int num = rndGenerator.nextInt();
      System.out.println ("A random int: " + num);
      System.out.print("Another one: " + rndGenerator.nextInt());
   }
}
```

`Random` is an example of a "singleton class" (usually)

- Unlike the classes we saw before, the `Random` constructor is typically called only once, resulting in a single object
- This object is then used to generate as many pseudorandom numbers as needed.

Classes that generate things: `Scanner`

Example:

```
import java.util.Scanner;
public class ScanDemo {
    public static void main (String args[]) {
         String text = "   1 2   red      blue";
         Scanner s = new Scanner(text);
         System.out.println(s.next());
         System.out.println(s.next());
         System.out.println(s.next());
         System.out.println(s.next());
         s.close();
    }
}
```

Output:

```
1
2
red
blue
```

- A `Scanner` breaks its input into tokens using a delimiter pattern, which by default matches whitespace
- In previous examples we used to initialize the `Scanner` on `in`, representing the keyboard. As we see, this is just one possibility.

Outline

- Static classes
  - `Math`
  - `Arrays` (later)
- Classes that represent types (examples)
  - `Fraction`
  - `bankAccount`
  - `Turtle`
  - `String`
- Classes that generate things (examples)
  - `Random`

(More kinds of classes – later in the course)

- Packages

Packages and the `import` command

Classes are often organized in *class libraries*, also called *packages*

```
import java.util.Random; // Gives access to Random's services
public class RandomNumbersDemo {
   public static void main (String[] args){
      Random rndGenerator = new Random();   // rather than java.util.Random
      // as before ...
   }
}
```

```
import java.util.*;   // Gives access to all the classes in this package
// typically used when you need to access 2 or more classes in the package.
```

Examples of widely used classes:

- `java.lang.Math`
- `java.lang.String`
- `java.lang.System`

The `java.lang` library is automatically imported into every class you write.

Java's standard class library

- A collection of ~3,000 classes, organized in ~200 packages, included in every Java implementation
- Examples: `java.lang, java.util, java.io, java.security, java.util.zip, java.net, ..` .

ERROR: undefined
OFFENDING COMMAND:

STACK: