

PIC 20A

The Basics of Java

David Hyde
UCLA Mathematics

Last edited: March 29, 2020

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

Variables

As in any programming language, Java has variables.

```
public class Test {  
    public static void main(String[] args) {  
        double d1 = 0.2;  
        double d2 = 0.5;  
        System.out.println(d1*d2);  
    }  
}
```

In Java, there are *primitive* and *reference* variables.

Variable names

Names are case-sensitive and must not conflict with reserved keywords.

Reserved keywords include: `abstract`, `class`, `continue`, `else`, `for`, `int`, `new`, `static`, etc.

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

Primitive data types

There are 8 primitive data types.

- ▶ byte: 1 byte integer
- ▶ short: 2 byte integer
- ▶ int: 4 byte integer
- ▶ long: 8 byte integer
- ▶ float: single-precision floating-point number
- ▶ double: double-precision floating-point number
- ▶ boolean: a Boolean (it's not spelled bool)
- ▶ char: 16-bit unicode character

<http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html>

Literals

Literals represent a fixed value.

- ▶ Integer literals:

```
int i1 = 100; //decimal
int i2 = 0x1abc; //hex
int i3 = 0b11011; //binary
long i3 = 1000000L; //L means long
```

- ▶ Boolean literals

```
boolean b1 = true;
boolean b2 = false;
```

Literals

► Floating-point literals

```
double d1 = 123.4;  
double d2 = 1.234e2; //in scientific notation  
float f1 = 123.4f; //f means float
```

► Character literals

```
char c1 = 'a';  
char c2 = '\\'; //apostrophe character
```

class String

The class String is not a primitive data type. However it receives special support by the language, so people often think of it as such. More on Strings later.

Variable initialization

Variables should be initialized before use.

```
public class Test {  
    public static void main(String[] args) {  
        int i; //i is declared  
        //System.out.println(i); //won't compile  
        i = 0;    //i is assigned/initialized  
        System.out.println(i);  
    }  
}
```

(Some variables will be automatically initialized to a reasonable default value. Don't rely on this behavior, as it's bad style, and save yourself from memorizing the precise rules for this.)

Conversions

A variable of one type is converted to another with

- ▶ casting conversion and
- ▶ other implicit conversions.

Casting conversion converts from type T1 to T2 with (T2) T1.

```
double d = ((double) 1) / ((double) 2);
```

Roughly speaking, *implicit conversion* converts from type T1 to T2 when a variable of type T1 is provided when T2 is expected.

```
double d = 5;
```

Operators

Operators (which are functions) like ++, +, &&, =, and -= are similar to those of C++.

```
public class Test {  
    public static void main(String[] args) {  
        int i = 2;  
        System.out.println(++i); //3  
        System.out.println(i>0); //true  
        System.out.println((i>0) || false); //true  
    }  
}
```

Basic I/O

Here's how take input from and output to (I/O) the command line.

```
//import class Scanner from Java API
import java.util.Scanner;

public class Test {
    public static void main(String[] args) {
        Scanner reader = new Scanner(System.in);
        int i = reader.nextInt(); //get int from user
        System.out.println(i); //print out the int
    }
}
```

More on I/O and Scanner later.

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

Control structures

Control structures like if-statements, for-loops, and while-loops are similar to those of C++.

```
public class Test {  
    public static void main(String[] args) {  
        int i = 2;  
        if (i > 0) {  
            System.out.println("i is positive");  
        }  
        else if (i < 0) {  
            System.out.println("i is negative");  
        }  
        else {  
            System.out.println("i is zero");  
        }  
    }  
}
```

<http://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html>

Omitting curly braces

When one statement follows a control structure, { and } can be omitted.

```
public class Test {  
    public static void main(String[] args) {  
        int i = 2;  
        if (i > 0)  
            System.out.println("i is positive");  
        else if (i < 0)  
            System.out.println("i is negative");  
        else  
            System.out.println("i is zero");  
        for(int j=i; j<5; j++)  
            System.out.println(j);  
    }  
}
```

Some consider this bad style (I disagree) but I'll use it in lecture slides to save space.

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

classes

You should be familiar with these basics concepts of classes:

- ▶ classes,
- ▶ data members,
- ▶ member functions,
- ▶ constructors, and
- ▶ public and private members.

Let's see how they're done in Java.

Fields

In Java, data members are called *fields*.

```
public class Complex {  
    public double real; //fields  
    public double imag;  
}
```

In Java, the first letter of class is capitalized by convention. The class `Complex` has 2 fields named `real` and `imag`.

Access modifiers

In Java there are 4 access modifiers: public, private, protected, and package-private (no explicit modifier). We'll later talk about protected and package-private later.

```
public class Complex { //public class? what?  
    public double real; //public make sense  
    public double imag;  
}
```

You should be familiar with public and private fields. We'll talk about public classes later.

Constructors

You can have multiple constructors.

```
public class Complex {  
    public double real;  
    public double imag;  
    public Complex() {  
        real = 0; imag = 0;  
    }  
    public Complex(double r, double i) {  
        real = r; imag = i;  
    }  
}
```

(You will soon see why private constructors are sometimes useful.)

Methods

In Java, member functions are called *methods*.

```
public class Complex {  
    public double real;  
    public double imag;  
    ...  
    public void printNum() {  
        System.out.println(real+" "+imag+"i");  
    }  
}
```

Reference variables

Let's see how to use a class in another class.

```
public class Test {  
    public static void main(String[] args) {  
        Complex c1; //c1 is a reference variable  
        c1 = new Complex(); //c1 refers to a Complex  
        Complex c2 = new Complex(0.1,1.2);  
    }  
}
```

In this example, `c1` and `c2` are *reference variables* of type `Complex`. You must use the `new` keyword to instantiate an Object. (In Java, primitive variables are not Objects.)

Reference variables are like pointers

c1 is not the Object. c1 refers to the Object.

```
public class Test {  
    public static void main(String[] args) {  
        Complex c1 = new Complex(1,2);  
        Complex c2 = c1; //copies the reference  
        c2.real = 10;  
        System.out.println(c1.real); //output 10.0  
    }  
}
```

(We'll later talk about how to copy Objects.)

Reference variables are like pointers

We can change what the reference variable refers to. With `null`, we can make it refer to nothing.

```
public class Test {  
    public static void main(String[] args) {  
        Complex c1 = new Complex(1,2);  
        c1 = new Complex(3,3); //first Complex gone  
        c1 = null; //second Complex gone  
    }  
}
```

(The garbage collector will automatically destroy Objects with no reference variables referring to it. More on this later.)

returning Objects

Methods can return references to Objects.

```
public class Complex {  
    public double real;  
    public double imag;  
    ...  
    public Complex(double r, double i) {  
        real = r; imag = i;  
    }  
    ...  
    public Complex add(Complex c) {  
        return new Complex(real+c.real, imag+c.imag);  
    }  
}
```

returning Objects

```
public class Test {  
    public static void main(String[] args) {  
        Complex c1 = new Complex(1,2);  
        Complex c2 = new Complex(3,4);  
        Complex c3 = c1.add(c2);  
        c3.printNum(); //output 4.0+6.0i  
  
        Complex c4 =  
            (new Complex(5,6)).add(new Complex(7,8));  
        c4.printNum(); //output 12.0+14.0i  
    }  
}
```

Package-private classes

- ▶ classes can be public or package-private (no explicit modifier).
- ▶ (Top-level) classes cannot be private or protected.
- ▶ Package-private classes can be accessed within the same file and same package. (More on packages soon.)
- ▶ If you know a class should not be use elsewhere, make it package-private.

```
public class Test {  
    public static void main(String[] args) {  
        SomeClass v1 = new SomeClass();  
        ...  
    }  
}  
  
class SomeClass {  
    ...  
}
```

Package-private fields and methods

- ▶ Fields and methods of a package-private class can be package-private.
- ▶ A public class can but probably shouldn't have package-private fields and methods.

public classes

- ▶ A public class can be used from other files. (class Complex should be public.)
- ▶ A file can have only one public class. It's name must match the filename.
- ▶ A public class can have a main function. The command
java ClassName
calls that main function.

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

Compiling multiple files

When you compile

```
javac Test.java
```

and `Test.java` uses a class named `Complex`, there are 3 scenarios:

1. `javac` finds and uses a package-private class `Complex` defined in the file `Test.java`.
2. `javac` finds and uses `Complex.java` or `Complex.class` in the current directory.
3. `javac` can't figure out what `Complex` is and issues an error.

(The option `-classpath` changes where to look for `Complex.java` or `Complex.class`.)

Compiling multiple files

When javac finds the definition of Complex in another file, there are 3 scenarios.

- ▶ There's `Complex.class` but not `Complex.java`. Then javac uses `Complex.class`.
- ▶ There's `Complex.java` but not `Complex.class`. Then javac compiles `Complex.java` and produce `Complex.class`.
- ▶ There's both `Complex.java` and `Complex.class`. If, according to the files' timestamps, `Complex.class` is newer than `Complex.java`, then `Complex.class` is used. If `Complex.java` is newer, it's recompiled and `Complex.class` is overwritten.

(The last rule says `Complex.java` is recompiled if it's been updated since the last compilation.)

Compiling multiple files

Summary: Put all `.java` files in the same directory, and explicitly compile only the file with the `main` function. The rules are set up so that things will just work.

We'll later talk about more sophisticated methods of managing more complicated code structures.

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

final variables

A final variable is can be set only once. It's like const in C++.

```
public static void main(String[] args) {  
    final int i; //blank final variable  
    int j = 2;  
    System.out.println(j);  
    i = 3;  
    System.out.println(i+j);  
    //i= 5; //error  
}
```

You do not have to initialize a final variable immediately.
An uninitialized final variable is called a *blank* final variable.

final parameters

You can make function parameters final to prevent certain bugs.

```
static void func(final int i, final Complex c) {  
    //i = 2; error  
    //c = new Complex(1,2); //error  
    c.real = 5; //not an error!  
}  
  
public static void main(String[] args) {  
    Complex c1 = new Complex(0,0);  
    func(7,c1);  
    c1.printNum(); //5.0+0.0i  
}
```

Pay attention to what's constant and what isn't.

final fields

final fields must be initialized by the end of the constructor.

```
class Complex {  
    final public double real, imag;  
    public Complex() {  
        real = 0; imag = 0;  
    }  
    public Complex(double r , double i) {  
        real = r ; imag = i ;  
    }  
}
```

Other uses of final

In addition to variables, methods and classes can be `final`. More on this when we talk about inheritance.

static fields

A static field belongs to the class, not the instantiated Object.
1 copy of it is shared across Objects of the same class.

```
public class Monster {  
    private static int count = 0; //initialize to 0  
    public Monster() {  
        count++;  
        System.out.print("New Monster created.");  
        System.out.print("There are ");  
        System.out.print(count + " Monsters.\n");  
    }  
}
```

static fields

```
public class Test {  
    public static void main(String[] args) {  
        Monster m1 = new Monster();  
        Monster m2 = new Monster();  
        Monster m3 = new Monster();  
    }  
}
```

The output is

New Monster created. There are 1 Monsters.

New Monster created. There are 2 Monsters.

New Monster created. There are 3 Monsters.

Constants with static fields

Use static final fields for constants.

```
public class MyMath {  
    public static final double PI = 3.141592653589;  
    public static final double E = 2.718281828450;  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        double r = 3;  
        System.out.println(2*MyMath.PI*r); //good  
        MyMath m1 = new MyMath();  
        System.out.println(2*m1.PI*r); //bad style  
    }  
}
```

static functions

A static function belongs to the class, not the instantiated Object. They can only access static fields.

```
public class Monster {  
    private static int count = 0;  
    private final String name;  
    ...  
    public static int countMonster() {  
        //error! Can't access non-static field  
        //System.out.println(name);  
        return count;  
    }  
}
```

static functions

Some static functions have nothing to do with objects, and are simply packaged together into a class.

```
public class MyMath {  
    public static final double PI = 3.141592653589;  
    public static final double E = 2.718281828450;  
    public static double sqrt(double d) { ... }  
    public static double sin(double d) { ... }  
    public static double cos(double d) { ... }  
    public static double tan(double d) { ... }  
}
```

Other uses of static

In addition to fields and methods, initialization blocks can be `static`.
More on this soon.

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

Arrays

An *array* is an Object of reference type which contains a fixed number of components of the same type.

```
double[] d_arr;
```

d_arr is a reference variable of type double[].

Few things to note.

- ▶ Length is fixed.
- ▶ The array length is not part of its type.
- ▶ Arrays use zero-based indexing.

Array creation expression

Create an array with an *array creation expression*.

```
int[] i_arr = new int[5];  
for (int i=0; i<5; i++)  
    i_arr[i] = 0;  
i_arr = new int[9];
```

(The length of i_arr is not part of its type.)

Array creation expression

Arrays of Objects have the same syntax.

```
Complex[] c_arr = new Complex[5];  
for (int i=0; i<5; i++)  
    c_arr[i] = new Complex();  
c_arr = null;
```

Let's parse this carefully.

Array creation expression

Create a reference variable `c_arr` of type `Complex[]`.

```
Complex[] c_arr;
```

Create an Object of type `Complex[]` that contains 5 reference variables of type `Complex`.

```
new Complex[5];
```

Make `c_arr` refer to the `Complex[]` Object.

```
c_arr = new Complex[5];
```

Array initialization

At this point, no Complex Objects exist. Create the Complex Objects.

```
for (int i=0; i<5; i++)  
    c_arr[i] = new Complex();
```

c_arr can be set to null (and we lose reference to the Complex[] Object and 5 Complex Objects).

```
c_arr = null;
```

Array initializer

We can also use the *array initializer*.

```
int[] i_arr = {1,2,3};  
//i_arr = {2,3,4,5,6}; //error!  
i_arr = new int[] {2,3,4,5,6};  
Complex[] c_arr = {new Complex(0,1), //(*)  
                   new Complex(), null};  
c_arr[2] = new Complex(3,8);
```

(Newline at (*) only because we're out of space.)

Let's parse this carefully.

Array initializer

When declaring and initializing an array at the same time, use the array initializer by itself.

```
int[] i_arr = {1,2,3};
```

Otherwise, use the array initializer as part of an array creation expression.

```
i_arr = new int[] {2,3,4,5,6};
```

Array initializer

Create an Object of type `Complex[]` with 3 `Complex` references. 2 `Complex` Objects created and the first 2 `Complex` references refer to them. The last `Complex` reference refers to nothing.

```
Complex[] c_arr = {new Complex(0,1),  
                   new Complex(), null};
```

Create another `Complex` Object and make the last `Complex` reference refer to it.

```
c_arr[2] = new Complex();
```

Array length

An array has a public final field named length.

```
int[] i_a = new int[300];  
for (int i=0; i<i_a.length; i++)  
    i_a[i] = 0;  
//i_a.length = 2; //error!
```

Varargs

A function can take a variable number of arguments (varargs)

```
static void fn(int i, double d, char... c_arr) {  
    System.out.println(i);  
    System.out.println(d);  
    for (int j=0; j<c_arr.length; j++)  
        System.out.println(c_arr[j]);  
}
```

fn can be called with 2, 3, or more arguments.

```
public static void main(String[] args) {  
    fn(1,2.2);  
    fn(5,1.2,'a','b','c','d');  
}
```

The variable number of inputs are converted into an array.
(Varargs can be used only in the final argument position.)

Varargs

You can also use an array as your input.

```
public static void main(String[] args) {  
    fn(1,2.2);  
    fn(5,1.2,'a','b','c','d');  
    fn(5,1.2, new char[] {'a','b','c'});  
}
```


Varargs

A varargs input allows variable inputs or an array.

An array input only allows an array.

```
static void fn(int i, double d, char[] c_arr) {  
    ...  
}  
...  
  
public static void main(String[] args) {  
    //fn(5,1.2,'a','b','c','d'); //error!  
    fn(5,1.2, new char[] {'a','b','c'});  
}
```

Enhanced for loop

In Java, there are two types of for loops: regular and *enhanced* for loops.

```
public static void main(String[] args) {  
    int[] i_arr = {1,5,0,2,2,9,1};  
    //regular for loop  
    for (int i=0; i<i_arr.length; i++)  
        System.out.println(i_arr[i]);  
    //enhanced for loop  
    for (int j : i_arr)  
        System.out.println(j);  
}
```

The enhanced for loop is also called the *for-each* loop.

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

Strings

`String` is an `Object` (not a primitive type) that represents a sequence of characters.

Strings have a lot of special support by the Java language such as `String` literals and the `+` operator.

Strings are *immutable*. To change a `String` you have to create a new one with the desired changes and discard the old one. The class `StringBuffer` provides a mutable array of characters.

Special String features

Use " to write String literals.

```
String s1 = "Good morning.";
```

Use the + operator to concatenate other Objects with strings.

Object + String

toString method

All Objects have a method with signature

```
public String toString()
```

whether you explicitly make one or not. This will be explained later in the context of inheritance.

The `System.out.println` method calls `toString` on its input.

```
Complex c1 = new Complex();  
System.out.println(c1);  
System.out.println(c1.toString());
```

The output is

```
Complex@15db9742  
Complex@15db9742
```

toString method

You can override the toString method

```
public class Complex {  
    ...  
    public String toString() {  
        return real + "+" + imag + "i";  
    }  
}
```

and you get a better output.

```
Complex c1 = new Complex(2,3);  
System.out.println(c1);  
System.out.println(c1.toString());
```

The output is

2+3i

2+3i

The + operator

The + operator on

`primitiveType + String`

(or vice versa) converts the primitive type to an appropriate `String` and concatenates it to the other `String`.

The + operator on

`Object + String`

calls the `toString` on the `Object` and concatenates the output to the other `String`.

Math, a utility class

A *utility class* contains a set of common and reused members. A utility class is more of a package rather than a blueprint for an Object.

```
public class Test {  
    public static void main(String[] args) {  
        System.out.println(Math.sin(0.1));  
        //sin is a static method of Math  
    }  
}
```

Utility classes use static members to provide their functionalities.

Math, a utility class

In fact, you are not allowed to instantiate Math.

```
public class Test {  
    public static void main(String[] args) {  
        Math m1 = new Math();  
    }  
}
```

Compiling this produces

```
error: Math() has private access in Math  
Math m1 = new Math();
```

More on this later.

Take a look at:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html>

main

The main function of a public class must have signature

```
public static void main(String[] args)
```

or

```
public static void main(String... args)
```

The parameter args contain the inputs provided through the command line

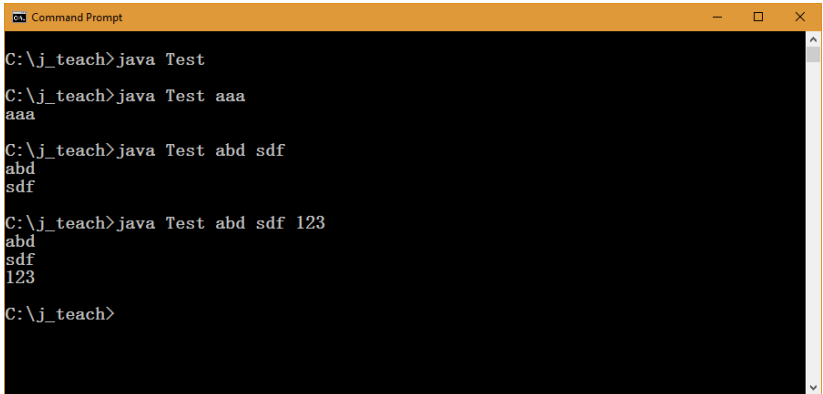
```
java ClassName input1 input2 ...
```

main

You can use args to allow users to specify certain options.

```
public class Test {  
    public static void main(String[] args) {  
        for (String s : args)  
            System.out.println(s);  
    }  
}
```

main



```
Command Prompt

C:\j_teach>java Test
C:\j_teach>java Test aaa
aaa
C:\j_teach>java Test abd sdf
abd
sdf
C:\j_teach>java Test abd sdf 123
abd
sdf
123
C:\j_teach>
```

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

packages and fully qualified names

Java code is organized through *packages* like `java.lang` or `java.util`. To use a class within a package, refer to its *fully qualified name*.

```
public class Test {  
    public static void main(String[] args) {  
        java.util.Scanner r; //fully qualified name  
        ...  
    }  
}
```

import declarations

When using a class within a package, it's convenient to import it.

"From the package `java.util` import the (public) class `Scanner`."

```
import java.util.Scanner;

public class Test {
    public static void main(String[] args) {
        Scanner r;
        ...
    }
}
```


import declarations

You can import an entire package.

“Import all classes from the package java.util.”

```
import java.util.*;

public class Test {
    public static void main(String[] args) {
        Scanner r;
        Calendar c;
        ...
    }
}
```

java.lang

The java.lang is special; all classes in it are automatically imported.

```
//import java.lang.*; //imported by default

public class Test {
    public static void main(String[] args) {
        java.lang.String s;
        double p = java.lang.Math.PI;
        double d = java.lang.Math.sqrt(p);
        java.lang.System.out.println(d);
    }
}
```

static imports

Import static fields and methods of a class with static imports.

```
import static java.lang.Math.PI; //static field
import static java.lang.Math.sqrt; //st. method
import static java.lang.System.out; //st. field

public class Test {
    public static void main(String[] args) {
        out.println(sqrt(PI));
    }
}
```

static imports

You can import all static members of a class.

```
import static java.lang.Math.*;

public class Test {
    public static void main(String[] args) {
        System.out.println(sqrt(PI));
    }
}
```

Are static imports bad?

Experts advise to use static imports *very sparingly*.

```
import static java.lang.Math.E;
import ...;
import ...;

public class Test {
    public static void main(String[] args) {
        //where does E come from?
        System.out.println(E);
    }
}
```

They make code less readable.

Are static imports bad?

When you static import from an entire class,

```
import static java.lang.Math.*;
```

you import too many names you don't know. (Do you know all the static members of Math?) This opens up the possibility of name conflicts and bugs.

Java API

The Java language comes with a library called the *Java API* (Application Program Interface).

The Java API is organized into packages like `java.lang`, `java.util`, `java.net`, `java.io`, or `javax.swing`.

Becoming proficient with Java involves learning key parts of the Java API.

Creating a package

Create a package with the package declaration.

```
package packageName;  
  
import java.util.Scanner;  
  
public class SomeClass {  
    ...  
}
```


Unnamed package

When no package name is specified, the class belongs to the *unnamed* package.

The unnamed package is for developing small or temporary applications. Generally, you should put all classes in named packages.

Managing source and class files

With respect to a base directory, place the files for a package named `packageName` within the folder `packageName`. Place the files for the unnamed package in the base directory.

For example,

```
package packageName;  
  
public class SomeClass {  
    ...  
}
```

this file should be placed in
`baseDirectory\packageName\SomeClass.java`

Managing source and class files

To compile a .java file within a package, use

```
javac packageName\Test.java
```

from the base directory.

To run the main function of a class within a package, use

```
java packageName.Test
```

from the base directory.

Managing source and class files

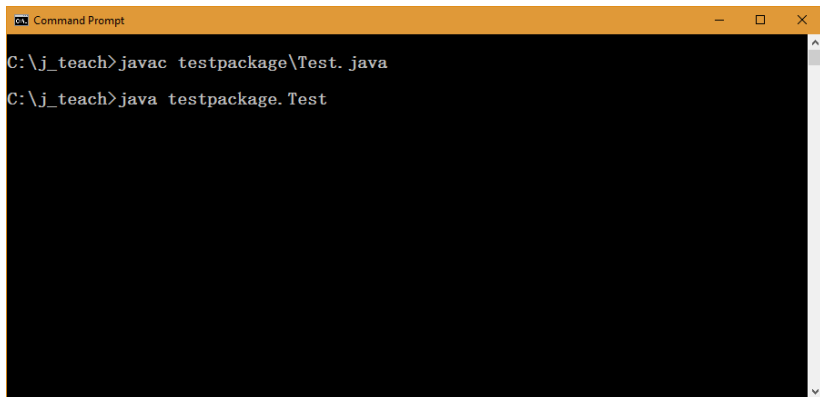
For example, to run the following main function,

```
package testpackage;

public class Test {
    public static void main(String[] args) {
        ...
    }
}
```

Managing source and class files

do this

A screenshot of a Windows Command Prompt window. The title bar is orange and says "Command Prompt". The window has standard Windows window controls (minimize, maximize, close) on the right. The background is black, and the text is white. Two lines of text are visible: "C:\j_teach>javac testpackage\Test.java" and "C:\j_teach>java testpackage.Test".

```
C:\j_teach>javac testpackage\Test.java
C:\j_teach>java testpackage.Test
```

(Or you can place the `main` function within a class within the unnamed package and save yourself from typing the package name.)

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

The Garbage Collector

Java periodically deletes unreferenced Objects. This process is called *garbage collection* and is performed by the *garbage collector*.

When the garbage collector runs is unpredictable and is none of your business. Sometimes unreferenced Objects may never be garbage collected (and the memory is returned to the OS as the program exists). Your program should not rely on the garbage collector behaving in a certain way.

However, you may want to explicitly control garbage collection for debugging purposes.

finalize

The garbage collector will run the `finalize()` method before destroying the `Object`.

Because garbage collection is unpredictable, you shouldn't rely on `finalize()` to clean up your `Object`. (Exceptions are rare.) So don't think of `finalize()` as the equivalent of destructors in C++. Only use `finalize()` as a debugging tool.

finalize

```
public class Monster {  
    private static int monster_count = 0;  
    private boolean isAlive;  
    public Monster() {  
        monster_count++;  
        isAlive = true;  
    }  
    public void kill() {  
        monster_count--;  
        isAlive = false;  
    }  
    protected void finalize() {  
        if (isAlive)  
            System.out.println("Live Monster is being"  
                               + " garbage collected.");  
    }  
}
```

System.gc()

The static method `System.gc()` *suggests* Java to run the garbage collector. This is only a suggestion, and it may be ignored.

```
public class Test {  
    public static void main(String[] args) {  
        Monster m1 = new Monster();  
        Monster m2 = new Monster();  
        m1.kill();  
        m1 = null; m2 = null;  
        System.gc(); //message from m2.finalize()  
    }  
}
```

Again, only use `System.gc()` as a debugging tool. If your code relies on `System.gc()`, then there's probably something wrong with it.

Outline

Variables

Control structures

classes

Compilation

final and static modifiers

Arrays

Examples: String, Math, and main

imports and packages

Garbage collection

More on constructors, this, and initializer blocks

Default constructor

When there are no constructors, the compiler provides a no-argument constructor called the *default constructor*.

```
public class Complex {  
    public double real, imag;  
    //the same no-argument constructor provided  
    //public Complex () {}  
}
```

The default constructor basically does nothing.

private constructor

Provide a single private constructor to prevent users from instantiating a class. This is a useful safety feature for utility classes.

```
public class MyMath {  
    ...  
    private MyMath() {}  
}
```

Copy constructor

By convention, the constructor of class T with a single input of type T is called the *copy constructor* and is used to copy Objects.

```
public class Complex {  
    public double real, imag;  
    ...  
    public Complex(Complex c) {  
        real = c.real; imag = c.imag;  
    }  
}
```

(In Java, there is no automatic default copy constructor.)

= doesn't copy Objects

Again, you can copy primitive variables but not reference variables with =.

```
public class Test {  
    public static void main(String[] args) {  
        int i1 = 7;  
        int i2 = i1; //copy  
        Complex c1 = new Complex(1,4);  
        Complex c2 = c1; //not a copy  
        Complex c3 = new Complex(c1); //copy  
    }  
}
```

The operator = copies the reference, so c1 and c2 refers to the same Object.

Don't copy immutable Objects

You almost never have to copy immutable Objects.

```
public class Test {  
    public static void main(String[] args) {  
        String s1 = "Hello";  
        String s2 = s1;  
        String s3 = new String(s1); //why?  
    }  
}
```


Factory methods

A *factory method* is a static method that returns an instance of the class.

```
public class Complex {  
    public double real, imag;  
    public static Complex fromPolar  
        (double r, double th) {  
        double re = r*Math.cos(th);  
        double im = r*Math.sin(th)  
        return new Complex(re,im);  
    }  
    ...  
}
```

Factory methods

There are 2 major advantages of factory methods.

- ▶ You can have multiple factory methods with the same number and types of inputs.
- ▶ The factory method's name can be descriptive.

```
public class Test {  
    public static void main(String[] args) {  
        Complex c1, c2;  
        c1 = Complex.fromXY(0.1, 3.1);  
        c2 = Complex.fromPolar(5.1, Math.PI/2);  
    }  
}
```

Copy factory

You can use a copy factory method instead of a copy constructor.

```
public class Complex {  
    public double real, imag;  
    public static Complex copy(Complex c) {  
        return new Complex(c.real, c.imag);  
    }  
    ...  
}
```

In my opinion, copy constructors and copy factories are both fine.

Forcing factory methods

Make all constructors private to force users to use factory methods.

```
public class Complex {  
    public double real, imag;  
    public static Complex fromPolar  
        (double r, double th) { ... }  
    public static Complex fromXY  
        (double real, double imag) {  
        return new Complex(real,imag);  
    }  
    private Complex(double r, double i) {  
        real = r; imag = i;  
    }  
}
```

this

Within a non-static method or a constructor, `this` is a reference to the current `Object`.

```
public class Complex {  
    public double real, imag;  
    ...  
    public Complex(double real, double imag) {  
        this.real = real; this.imag = imag;  
    }  
}
```

This is the most common use of `this`, to access a field which name is shadowed by a method or constructor parameter.

this for explicit constructor invocation

Another use of `this` is for *explicit constructor invocation*. This is also called *constructor chaining*.

```
public class Complex {  
    public double real, imag;  
    public Complex() {  
        this(0,0);  
    }  
    public Complex(double real, double imag) {  
        this.real = real; this.imag = imag;  
    }  
}
```

When you use `this` to call another constructor, you must do so in the first line of the constructor.

Initializer block

You can use a *initializer block* to avoid repeating code in constructors.

```
import java.util.Date;

public class Monster {
    private String dateOfBirth;
    private String name;
    {
        dateOfBirth = (new Date()).toString();
    }
    public Monster() {
        this("Unnamed");
    }
    public Monster(String name) {
        this.name = name;
    }
}
```

Initializer block

The initializer block is also called an *instance initializer block*, in contrast with the `static` initializer block.

The initializer block is run every time the `class` is instantiated, and it is run right before the constructor is run.

Initializer blocks aren't strictly necessary. You can achieve the same effect with a private method

Static initializer block

The *static initializer block* is run once when the class is loaded.

```
public class Card {  
    private static final String[] suits;  
    private static final char[] nums;  
    static {  
        suits = new String[]  
            {"Clubs", "Diamonds", "Hearts", "Spades"};  
        nums = new char[13];  
        nums[0] = 'A'; nums[10] = 'J';  
        nums[11] = 'Q'; nums[12] = 'K';  
        for (int i=1; i<=9; i++)  
            nums[i] = ( (char) ('0' + i) );  
    }  
    ...  
}
```

static initializer blocks are mostly used to initialize static fields.

Initializing fields

You can also initialize a field in its declaration.

```
public class Complex {  
    public double real = 0, imag = 0;  
}
```

The right-hand side can be a function evaluation.

```
import java.util.Date;  
  
public class Monster {  
    private final String dateOfBirth  
        = (new Date()).toString();  
    ...  
}
```

Initializing fields

There are 2 ways to initialize a `final static` field:

- ▶ in its declaration or
- ▶ in a static initialization block.

There are 3 ways to initialize a `final non-static` field:

- ▶ in its declaration,
- ▶ in an initialization block, or
- ▶ in a constructor.

There is more than one way to initialize a field. Use whichever is convenient for the setting.