Computer Systems Design and Applications

CS209A

Lecture 3

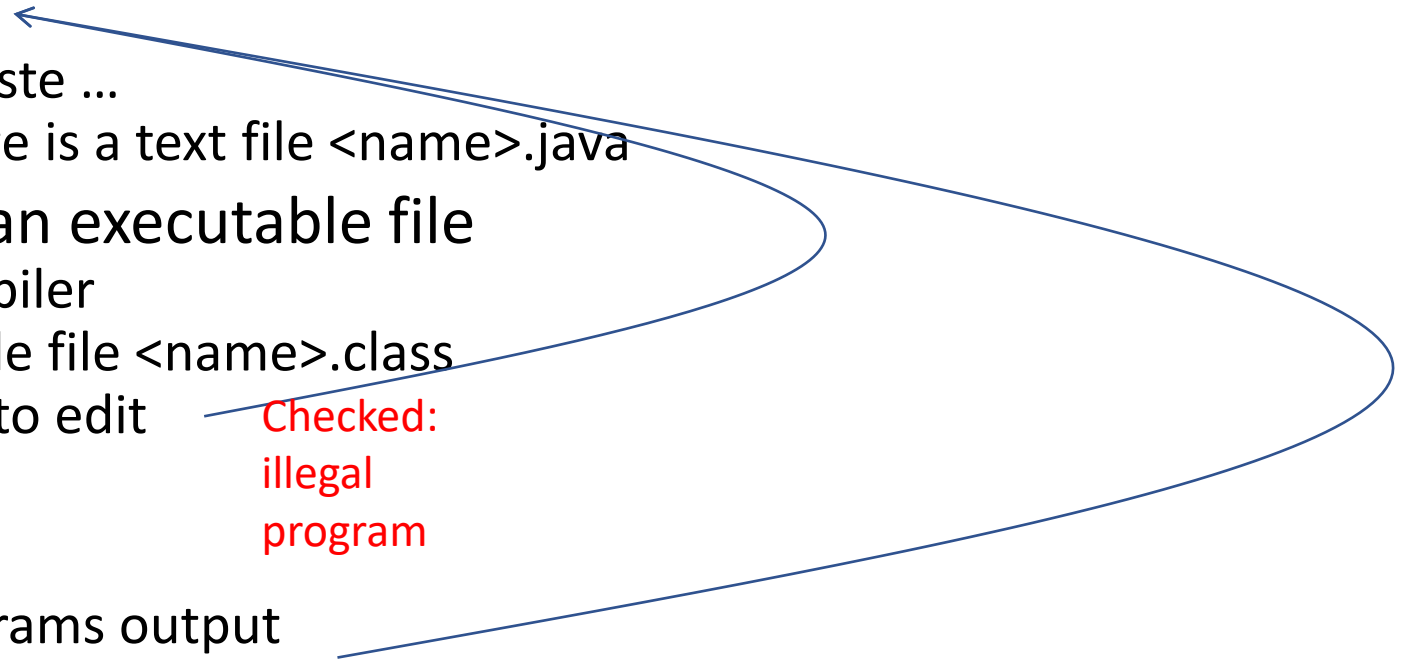
Adam Ghandar

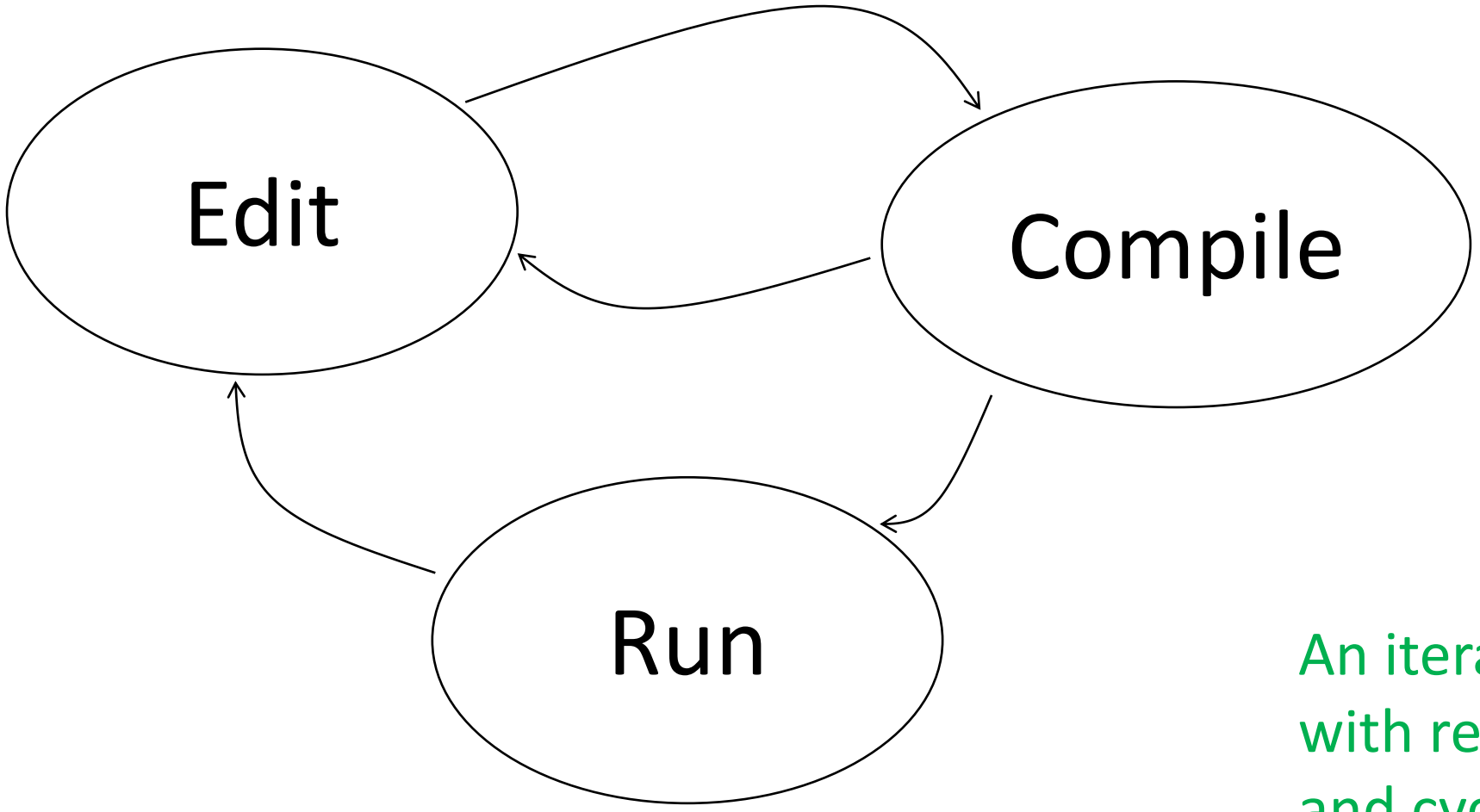
This week

- Process of developing software programs – history and now
- Types
- ADTs
- Generics
- Collections

Making a program

- A three step process *with feedback*
- **Edit** your program
  - Type it, cut and paste …
  - Output of this stage is a text file <name>.java
- **Compile** to create an executable file
  - Uses the java compiler
  - Output: a byte code file <name>.class
  - Mistake? Go back to edit
- **Run** the program
  - Uses the JRE
  - Output: your programs output
  - Mistake? Go back to edit

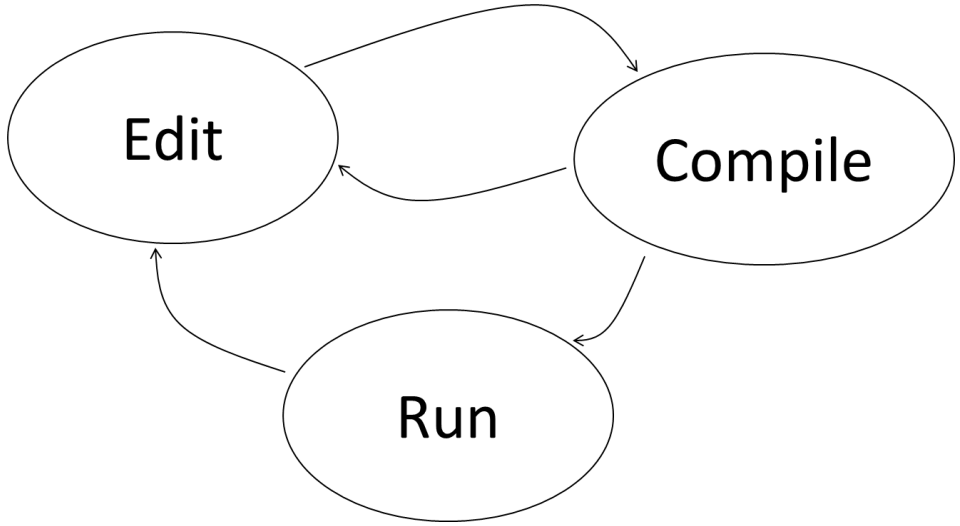


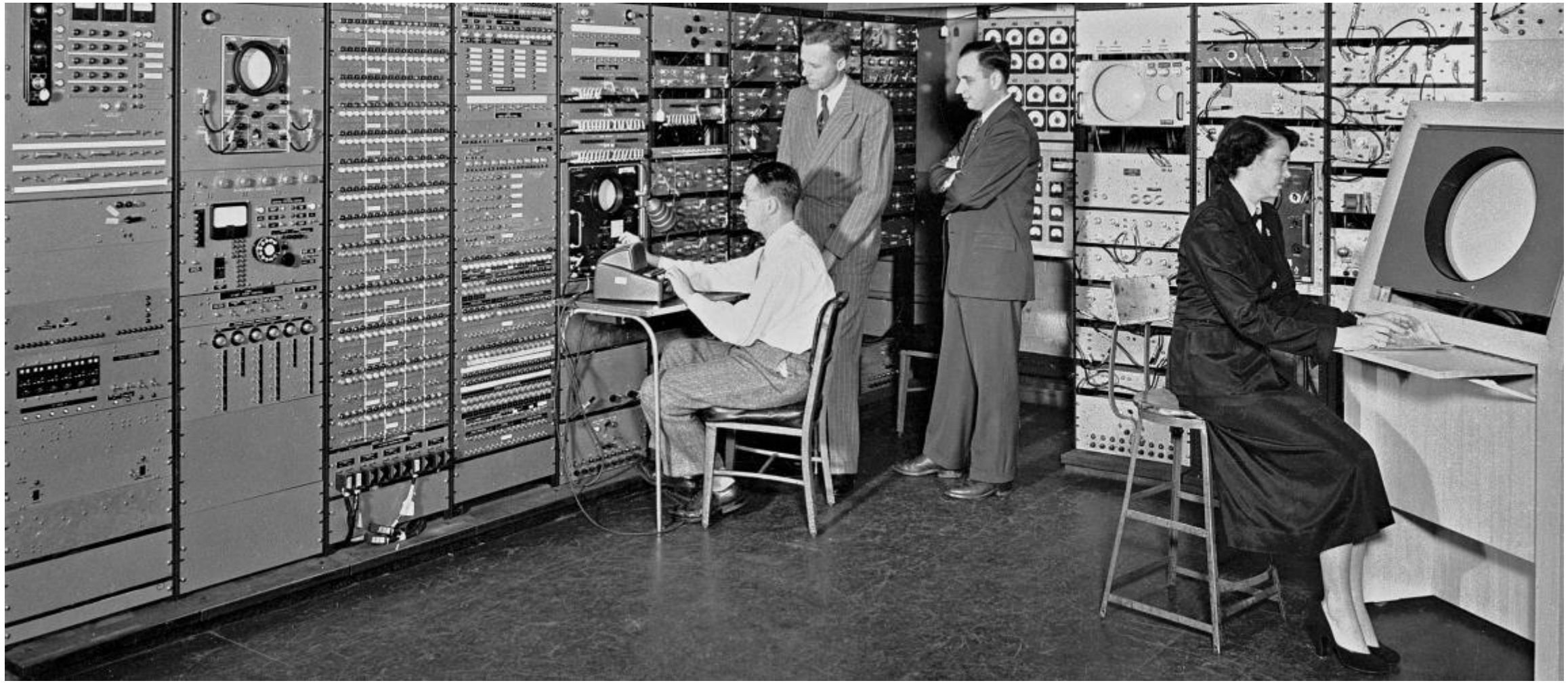


An iterative process with refinement and cyclic development

There are various software that support program development

- Virtual terminals
  - Use a text editor and command line
  - Can be good for beginners and advanced programmers
  - Simple and concise
- Integrated development environment (IDE)
  - Language system specific
  - Can be helpful for beginers
  - Useful tools included

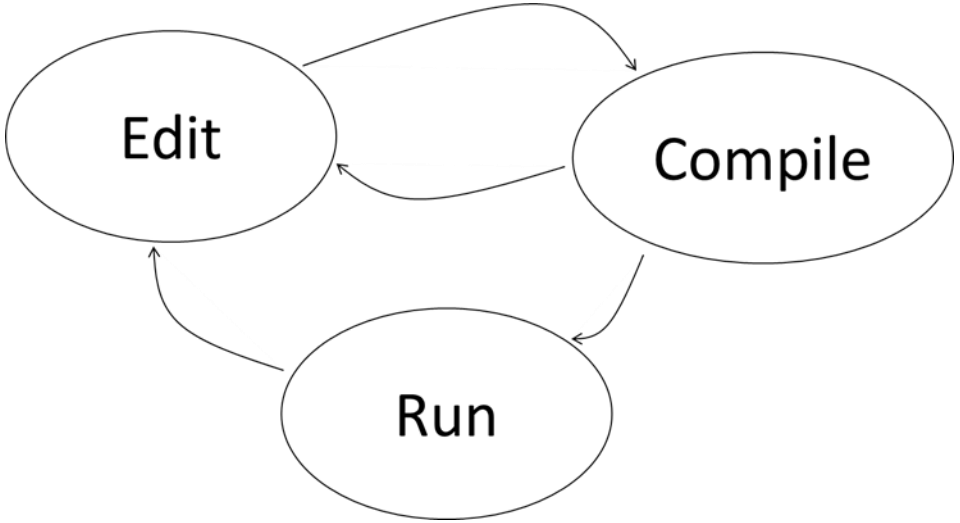




- Early computers used different methods for editing, compiling and running such as punch cards.
- Punch cards are much more efficient than even earlier systems with switches.

Punch cards were a common input method in the 1950s to early 1970s.

Edit – Compile – Run



Timeshare and terminals

Computer sharing and terminals

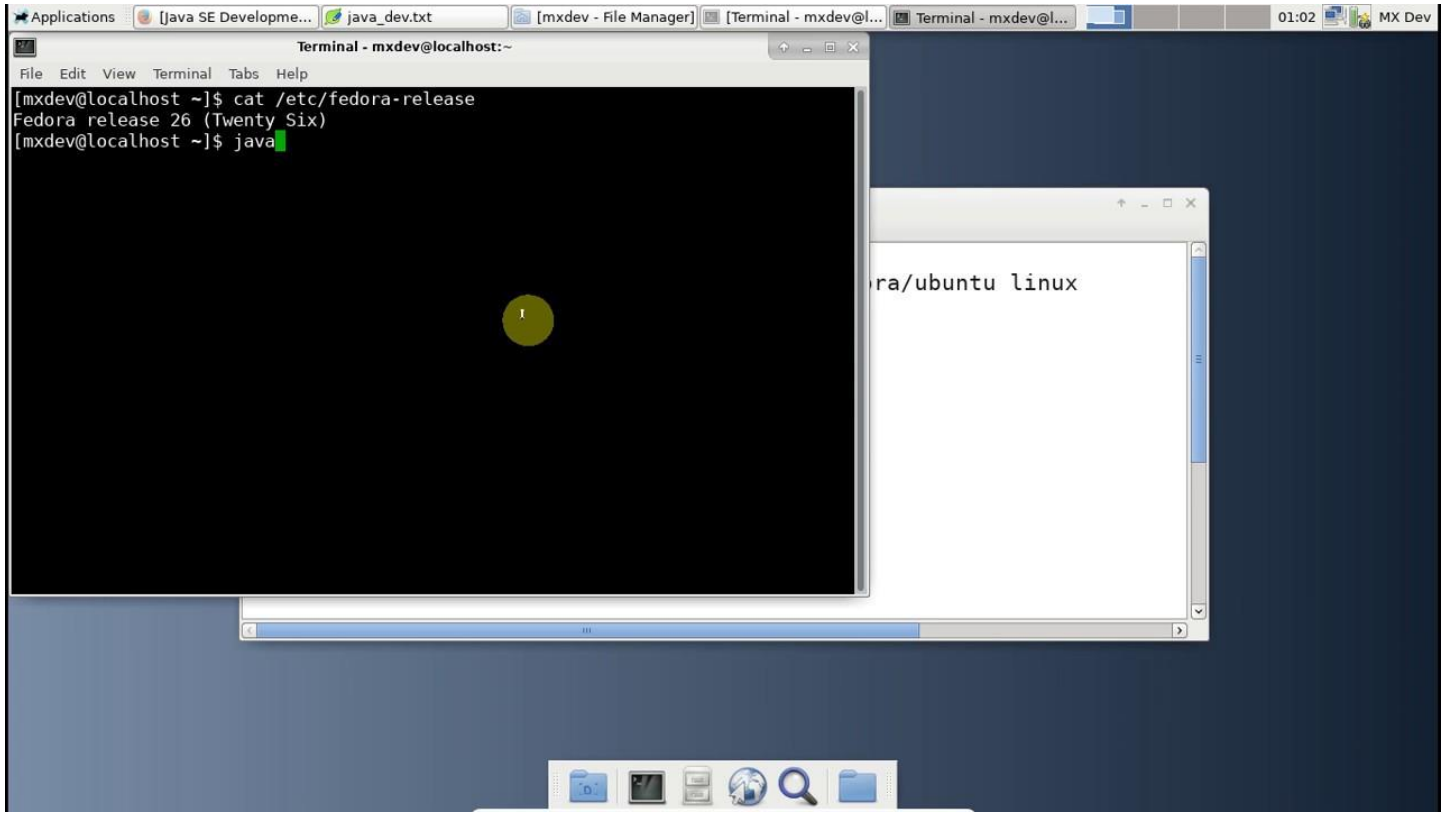
Virtual terminals – from 1980s

The PC processing model involves desktop/laptops with much more processing power than terminals

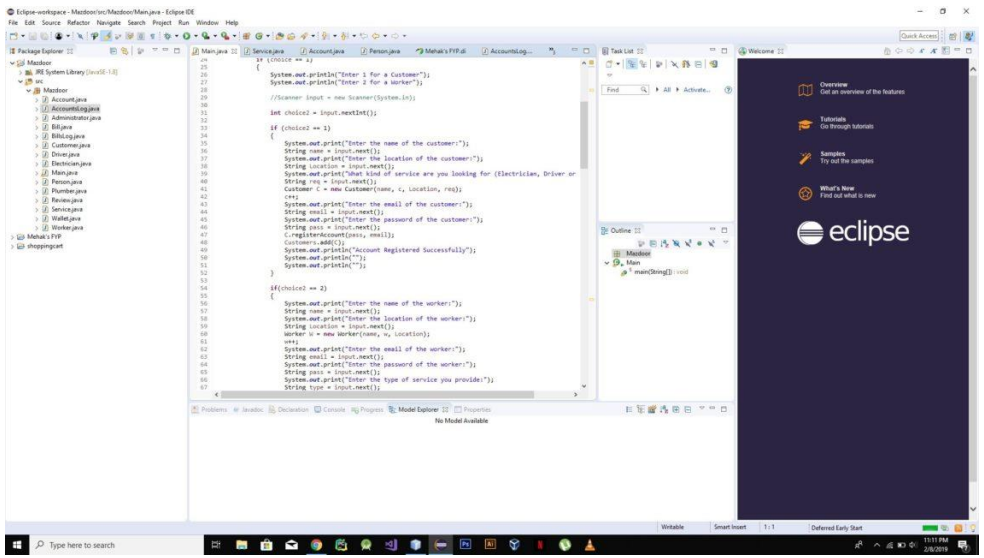
Virtual terminals emulate the idea of a terminal to interact with a computer CPU, you edit the program using a text editor in a virtual terminal

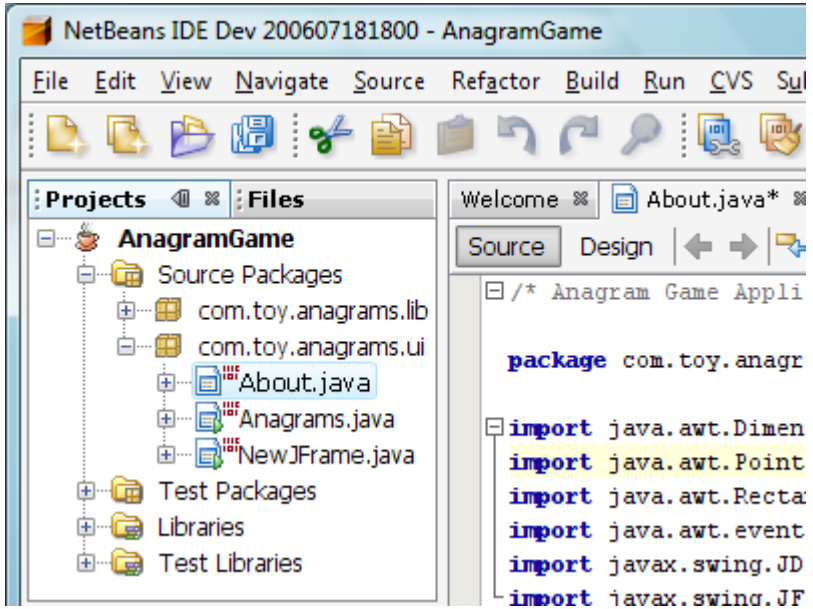
Window to run, window to edit, window to compile

Still used today



Integrated development environment





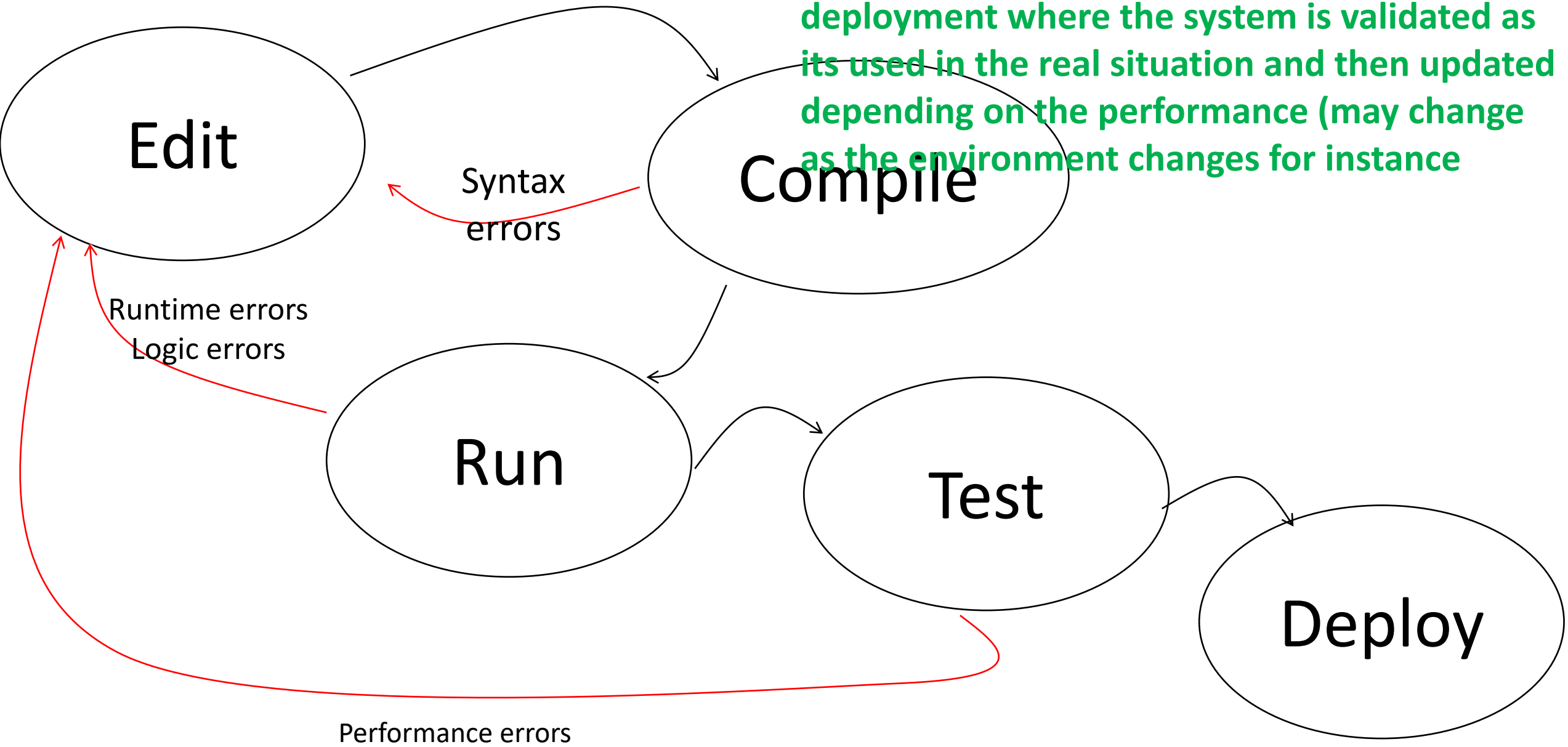
Also used since 1980s, IDEs provided an interface that can be more convenient. Many professionals use terminals but others use IDEs. Main benefit of IDEs is language specific tools and they are also used by professionals. They can provide a common platform such as is necessary in a company. Also IDEs can help with managing larger programs by providing hierarchy, searching.

Performance

Once we have iterated through the edit compile run cycle enough to get a program we can **test** (check/validate) and **deploy** (put to use in a real problem environment) our program.

In large scale projects it is more complex to ensure the program quality and speed of development (more code, more modules coupled together, and, larger teams). This makes it necessary to us approaches to manage the software engineering such as **agile development, test driven development** and **multiple deployment cycles**.

We can also add another iteration loop after deployment where the system is validated as its used in the real situation and then updated depending on the performance (may change as the environment changes for instance



Performance

- Predict program behaviour
  - When (or if) it will it finish?
- Compare algorithms and implementations
  - Will this change make my program faster?
  - How can I make my program faster
- Develop a basis to understand the problem and design new algorithms
  - Enables new technology
  - Enables new research
- We use empirical and mathematical analysis in computer science

Last Lecture: **RECURSION**

It is very useful for processing structured COLLECTIONS of OBJECTS

It is very useful for processing structured COLLECTIONS of OBJECTS

Recursion is very important with complex collections of objects

As we have seen with quick-sort: using recursion inside an array you can find smaller sub-arrays. Inside a set you reduce to smaller subsets. And when you divide enough, eventually you get empty or single element sets that can be solved trivially.

Before we get to collections we will learn about Types and Generics. Essentially an important way to reuse code that is part of the Java programming language.

You probably have already seen ArrayList, this is an example of generics.

Shifty type

TYPES and GENERICS in JAVA

Data types

A datatype is a set of values and a set of operations on those values

Data Type = { Objects } ∪ { Operations }

There are actually only
2^32 different int values

int = { 0, ±1, ±2, · · ·, -2^31, 2^31-1 }
∪ { +, −, ×, ÷, %, · · · }

Built in data types

- Char
  - Characters
  - Operations: eg compare
- String
  - Sequences of characters
  - Operations: eg concatenate

- Int
- Float
- Double
- Long
- Boolean
  - Truth values true, false
  - And, Or, Not, Xor

Remember: a data type is a set of values and a set of operations on those values

Boolean

- Boolean operations are used intensively to control the flow of programs
- Not, And, Or

p	$\neg p$
T	F
F	T

p	q	$p \wedge q$	$p \vee q$
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	F

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

T – true, F – false
And - &&
Or - ||
Not - !

Question: how to write A Xor B?
(!A && B) || (A && ! B)

Java Math Library

Class Math

- java.lang.Object
  - java.lang.Math

static double	**abs**(double a)Returns the absolute value of a double value.
static float	**abs**(float a)Returns the absolute value of a float value.
static int	**abs**(int a)Returns the absolute value of an int value.
static long	**abs**(long a)Returns the absolute value of a long value.

static double	**min**(double a, double b)Returns the smaller of two double values.
static float	**min**(float a, float b)Returns the smaller of two float values.

static double	**floor**(double a)Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
static int	**floorDiv**(int x, int y)Returns the largest (closest to positive infinity) int value that is less than or equal to the algebraic quotient.
static long	**floorDiv**(long x, long y)Returns the largest (closest to positive infinity) long value that is less than or equal to the algebraic quotient.

Variables and literals

- A **variable** is a name that refers to a value
- A **literal** is a programming language representation of a value
- Declarations
- Assignments

When we declare a variable we specify the values it can take and the operations that can be performed.

JavaExample.java

```
1  package com.beginnersbook;
2  import java.util.Scanner;
3  public class JavaExample
4  {
5      public static void main(String args[])
6      {
7          float p, r, t, sinterest;
8          Scanner scan = new Scanner(System.in);
9          System.out.print("Enter the Principal : ");
10         p = scan.nextFloat();
11         System.out.print("Enter the Rate of interest : ");
12         r = scan.nextFloat();
13         System.out.print("Enter the Time period : ");
14         t = scan.nextFloat();
15         scan.close();
16         sinterest = (p * r * t) / 100;
17         System.out.print("Simple erest is: " +sinterest);
18     }
19 }
```

Declare

Assign

literal

Problems | Javadoc | Declaration | Console | Progress | Cov

<terminated> JavaExample [Java Application] /Library/Java/JavaVirtualMachines/jdk-

```
Enter the Principal : 2000
Enter the Rate of interest : 6
Enter the Time period : 3
Simple Interest is: 360.0
```

Type conversion

- People like to use strings
- Computers use numbers

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

- Suppose you read an integer
- We need to call a system method Integer.parseInt() to convert the string to an integer
- This is **type conversion**

Type conversion

```
public class IntOps {

    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        int sum  = a + b;
        int prod = a * b;
        int quot = a / b;
        int rem  = a % b;

        System.out.println(a + " + " + b + " = " + sum);
        System.out.println(a + " * " + b + " = " + prod);
        System.out.println(a + " / " + b + " = " + quot);
        System.out.println(a + " % " + b + " = " + rem);
        System.out.println(a + " = " + quot + " * " + b + " + " + rem);
    }
}
```

Java converts the int to a string in the string concat operation

Type Checking

Types are checked in java. The java compiler is your comrade: it checks for type errors at compile time.

```java
public class Demo
{
    public static void main(String[] args)
    {
        String s = "12" * 2;
    }
}
```

```
$ java Demo.java
Demo.java:6: error: bad operand types for binary operator '*'
                String s = "12" * 2;
                           ^
  first type:  String
  second type: int
1 error
error: compilation failed
(base)
adamg@LAPTOP-HKEDU5DE ~/teaching
```

But often we want to convert a value from one type to another so they match

With built in types type conversion is easy…

- It even often happens automatically…

- To a string:
  - “x: 10” + 99 is converted to a String with the value “x: 99”

- Between numeric types they match if there is no loss of precision
  - int and floating point operands in 11 * 0.25 produces a double value 2.75

With built in types type conversion is easy…

- We can use special function calls in Double, Integer, Long, Boolean classes or other library calls

- Integer.parseInt(“123”) converts the String to an int
- Double.toString(10 * 0.23) converts the double to a String
- Math.round(2.712343) creates a long value 3

With built in types type conversion is easy…

- We can do casting…

- Cast a double to an int:

(int) 2.712343

This returns an int type by truncating the double value. It has a value of 2.

- Other examples:

(int) Math.round(2.712343) returns an int with value 3

11 * (int) 0.5 returns an int with value 0

As the last example shows we need to be careful of the type…


-00:00:17

Although type casting is easy…

It often causes bugs.

Short, int, float, double, long

- Why does java have different numeric types?
- Trade off in memory usage and range for integers (short vs int vs long)
- Trade off in memory use and precision for floating point values (double vs float)

	DATA TYPE	RANGE OF VALID VALUES	MEMORY VOLUME
integer	byte	from -128 to 127	1 byte
	short	from -32768 to 32767	2 bytes
	int	from -2147483648 to 2147483647	4 bytes
	long	from -9223372036854775808 to 9223372036854775807	8 bytes
floating point	float	from -3.4E + 38 to 3.4E + 38	4 bytes
	double	from -1.7E + 308 to 1.7E + 308	8 bytes
symbols	char	from 0 to 65536	2 bytes
logical	boolean	true or false	For value of this type 1 bit is enough, but in reality memory isn't provided by such portions, so variables of this type may be packed by virtual machine in different ways

Some conversions are impossible

(short) 70000

This 70000 is too big for a short which only can hold values up to 32767

What to do? Avoid doing it? Crash?

In java there is a well defined result

4464

Narrowing primitive conversion

It will not result in a runtime exception or other error

> A narrowing conversion of a signed integer to an integral type T simply discards all but the *n* lowest order bits, where *n* is the number of bits used to represent type T. In addition to a possible loss of information about the magnitude of the numeric value, this may cause the sign of the resulting value to differ from the sign of the input value. https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html

This may sometimes lead to loss of information depending on the situation.

Example of type conversion put to good use: pseudo-random integers

System method `Math.random()` returns a pseudo-random `double` value in [0, 1).

Problem: Given *N*, generate a pseudo-random *integer* between 0 and *N*−1.

```
public class RandomInt
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      double r = Math.random();
      int t = (int) (r * N);
      System.out.println(t);
   }
}
```

String to int (system method)

double to int (cast)

int to double (automatic)

```
% java RandomInt 6
3

% java RandomInt 6
0

% java RandomInt 10000
3184
```

Remember

A data type is a set of values and a set of operations on those values.

In java using types involves:

- Declaring the type
- Converting between types
- Finding and resolving type errors to compile your code: the compiler helps you to identify and fix type errors
- Pay attention to the type of your data

Strong typing and dynamic typing

Most languages that have the ambition of being used in critical applications insist on "strong typing". It means that the compiler will NOT let you mix variables of different types, unless they are known to be compatible (such as int and float). Note that many scripting languages take the opposite approach and guess the type of variables from how you are using them. It protects against unwanted effects We can also distinguish between static and dynamic typing approaches: in static **type** checking is done at compile-time, whereas **dynamic typed** languages are those in which **type** checking is done at run-time.

Java is statically typed but python is dynamically typed.

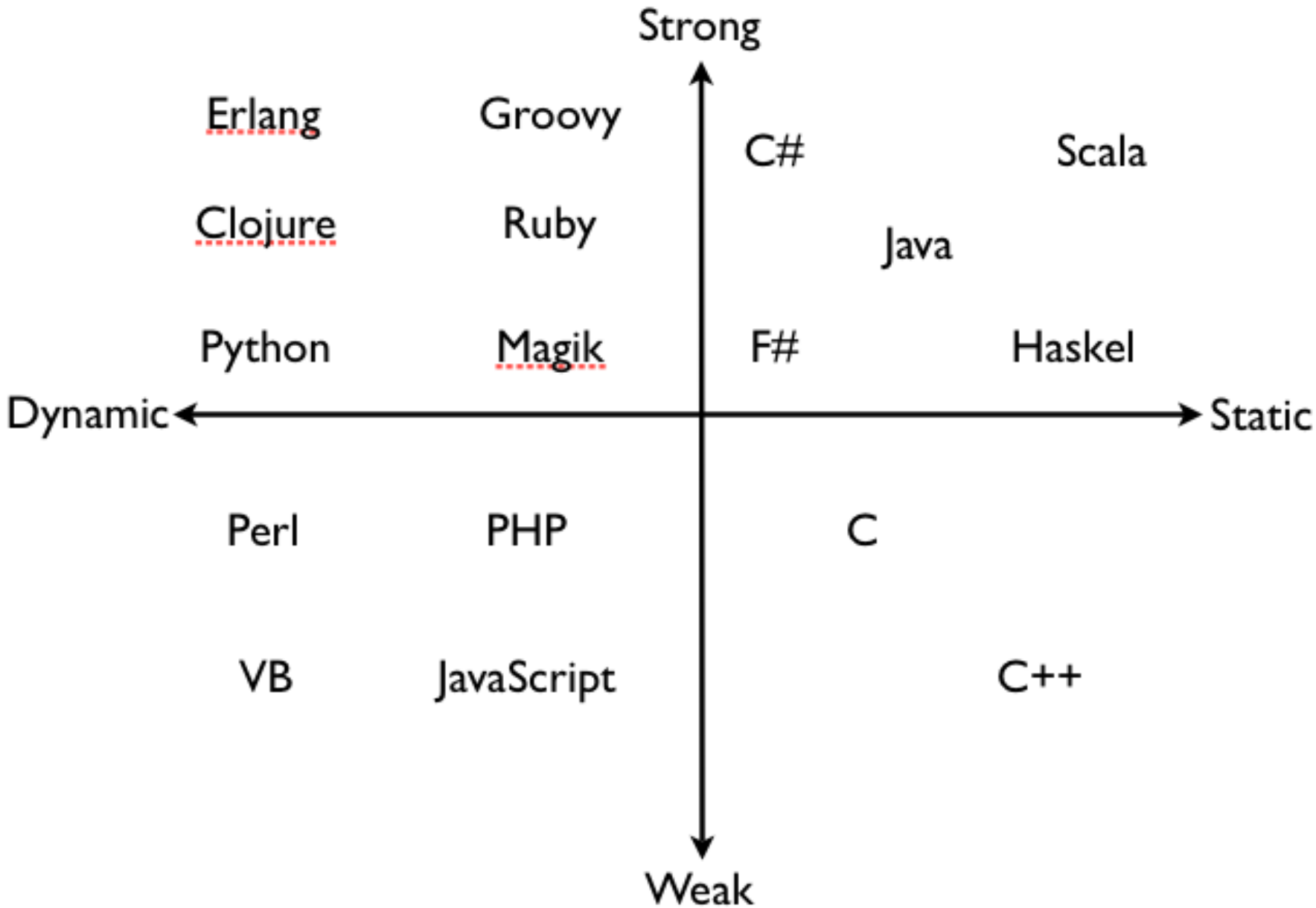
Strong Typing and Weak Typing

Strong Typing

- It means that the compiler will NOT let you mix variables of different types, unless they are known to be compatible (such as int and float)
- It protects against unwanted effects
- Java is strongly typed

Weak Typing

- Many scripting languages take the opposite approach and guess the type of variables from how you are using them
- Perl is an example of a weak typing


Strong
Erlang
Groovy
C#
Scala
Clojure
Ruby
Java
Python
Magik
F#
Haskel
Dynamic
Static
Perl
PHP
C
VB
JavaScript
C++
Weak


I LIKE THE STRONG SILENT TYPE

Java is strong typed, But…

```
4
5       String[] months = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
6               "Sep", "Oct", "Nov", "Dec"};
7
8       int[] sales = {120, 98, 75, 110, 150,
9               180, 170, 174};
10
```

```
Jan     Feb     Mar     Apr     May     Jun     Jul     Aug     Sep     Oct     Nov     Dec
120     98      75      110     150     180     170     174
```

One problem with **strong typing** is that it can lead to unnecessary issues, for instance if you want to write methods that display elements from an array, the same method cannot handle an array of Strings and an array of ints.

Java is strong typed, But…

```
15
16      public static void displayArr(String[] arr) {
17          int n = arr.length;
18          for (int i = 0; i < n; i++) {
19              if (i > 0) {
20                  System.out.print("\t");
21              }
22              System.out.print(arr[i]);
23          }
24          System.out.println("");
25      }
```

If this is the function that displays an array of Strings …

```
27      public static void displayArr(int[] arr) {
28          int n = arr.length;
29          for (int i = 0; i < n; i++) {
30              if (i > 0) {
31                  System.out.print("\t");
32              }
33              System.out.print(arr[i]);
34          }
35          System.out.println("");
```

Java is strong typed,
But…

- … you must overload it to display an array of integers. Note that, apart from the parameter type
- The code is identical except for the parameter
- Called overloading

Overloading

- Overloading allows you to give the same name to several functions. By looking at parameters, Java will **know** which one to call
- Different versions of a function
  - Same name
  - Same return type
  - Different parameters

Method

Signature

- The linker of the class loader that will match your code to methods with the same name that are available
- What defines the "signature" of a method?
  - The number of parameters
  - And their types

```
substring(int beginIndex)
```

Returns a new string that is a substring of this

```
substring(int beginIndex, int endI
```

Returns a new string that is a substring of this

isGreater(double, double)

isGreater(String, String)

isGreater(Date, Date)

```java
public class Overloading {

static int function(int n) {
    System.out.println("This is function(int)");
    return 0;
}
static int function(double x) {
    System.out.println("This is function(double)");
    return 0;
}

    public static void main(String[] args) {

        int n = 1;
        double val = 0;
        float f = 0;

        n = function(n);
        n = function(val);
        n = function(f);
    }
}
```

Output:

```
This is function(int)
This is function(double)
This is function(double)
```

In this example, when we call the function with a float, it's automatically "upgraded" to double and the function for doubles is called (the same function would be called for the int if there were no special function for integers)

Waste of time.

- If overloading helps keep the code safe, it's a waste of time to write identical code several times (even if we copy and paste).
- Even worse, if we want to change the code later, for instance to separate array elements by semicolons (;) instead of tabs when we print them, we must modify every method.

OMG: Code that works
With Strings and integers!!!

GENERICS: Computer-aided Overloading

- "Generics" comes from a **Latin** word that means "family" or "kind".
- It’s a sort of automatic overloading.
- It works using object references in Java.

IMPORTANT: Generics only works with OBJECTS

BEWARE: in Java, generics ONLY WORK with objects (references). Base variables, such as int, float, char, boolean variables ARE NOT objects. However, all base types have a "shadow" corresponding class (Integer, Float, Character, Boolean ...). For using generics, we must use these classes, which convert automatically to and from base data types (operations known as "boxing" and "unboxing")

```
3
4
5       String[] months = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
6               "Sep", "Oct", "Nov", "Dec"};
7
8       Integer[] sales = {120, 98, 75, 110, 150,180, 170, 174};
9
```

If we have an array of Strings (which are objects) and an array of Integers (that are objects too), then we can create a single method that will automatically work for both, without any having to write additional code for doing the overloading by having different overloaded methods…

```
Jan     Feb     Mar     Apr     May     Jun     Jul     Aug     Sep     Oct     Nov     Dec
120     98      75      110     150     180     170     174
```

Generic displayArray

```
3
4
5       String[] months = {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug",
6               "Sep", "Oct", "Nov", "Dec"};
7
8       Integer[] sales = {120, 98, 75, 110, 150,180, 170, 174};
9
```

```
36
37      static <T> void displayArray(T[] arr) {
38          int n = arr.length;
39
40          for (int i = 0; i < n; i++) {
41              if (i > 0) {
42              System.out.print("\t");
43              }
44              System.out.print(arr[i]);
45              }
46          System.out.println("");
47      }
48
49      public static void main(String args []) {
50          GenericDemo d = new GenericDemo();
51          displayArray(d.months) ;
52          displayArray(d.sales) ;
53      }
54  }
55
56
```

Before the return type of the method, you specify one or more symbols between angular brackets that represent Classes.

Only **Objects** are to be passed to the generic method

If you need several generic classes (for instance one for the return type and one for the parameter, or because you want to pass parameters from two different classes), you separate them with commas.

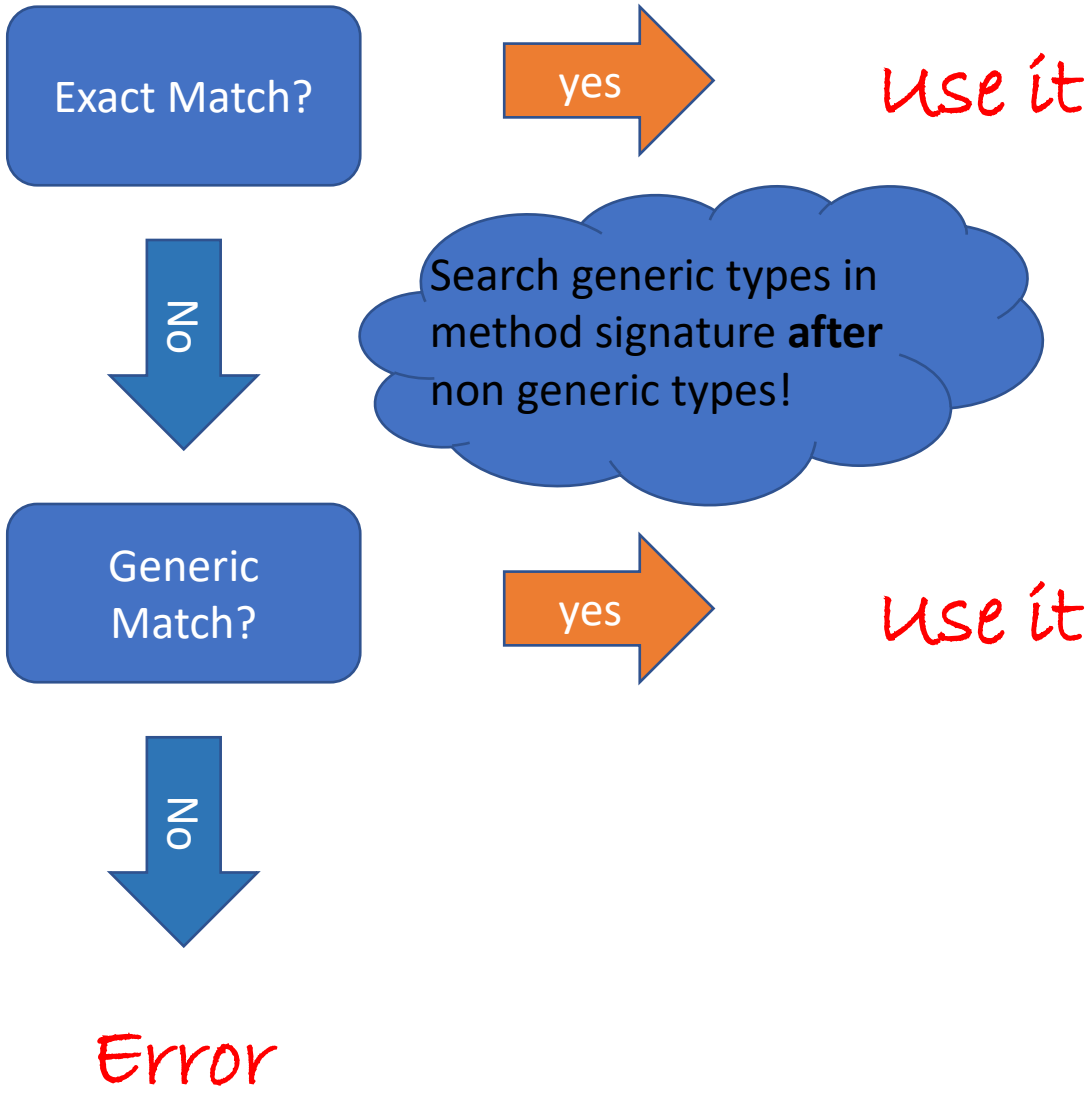
<T,U>

Generic methods can be overloaded

```
public static <T> void displayArray(T[] arr);

public static <T> void displayArray(T[] arr, String sep);

public static void displayArray(Date[] arr, String format);
```

Like any method, generic methods can also be overloaded, such as by another generic method or to handle a special case.

Example: Dates here which need a format to display.



The linker in the Class Loader Subsystem will try to find the best match.

If no match is found a type checking error is generated.

Everything can be generic.
Methods can be generic.
Even classes can be generic.

Important for collections!

A class can be used as a parameter. Collections are "container classes". For instance, an ArrayList is a class that can hold (contain) objects of any class.

Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

```java
public class Box<T> {
        private T t;

        public void add(T t) {
           this.t = t;
        }

        public T get() {
           return t;
        }

        public static void main(String[] args) {
                Box<Integer> integerBox = new Box<Integer>();
                Box<String> stringBox = new Box<String>();

                integerBox.add(new Integer(10));
                stringBox.add(new String("Hello World"));

                System.out.printf("Integer Value :%d\n\n", integerBox.get());
                System.out.printf("String Value :%s\n", stringBox.get());
            }
}
```

```
<terminated> Box [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home/bin/ja
Integer Value :10

String Value :Hello World
```

<terminated> Box [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home/bin/java (27 Feb 2020, 2:00:30 pm)

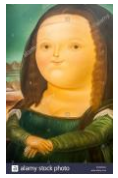
Data types that are objects

Important: a datatype is a set of values and a set of operations on those values

This is called an abstract data type (ADT) and is a very general concept

- Objects hold data type values and specify operations (methods)
- Examples:

Data type	Set of values	Operations examples
Color	3 8 bit integers	Get red, brighten
Picture	2D array of colors	Get/set color of a pixel (I,j)
String	Sequence of chars	Length, substring, compare



C A D T D

Abstract data types

- The concept of an ADT extends separating values and operations to objects that are more complex types
- **When we use abstract data types the representation is hidden from the client**
- They don’t need to know how bit strings that are colors are represented internally or how a picture is stored (is it a 2D array or a 1D array), or how data is stored in a server (is it in a database or a filesystem?)
- Clients can use ADTs without knowing implementation details
- This is useful, eg java could change the way strings are represented and everything written before would still work

Abstract Data Type in java

- To use a data type we need
  - Its name
  - How to construct new objects
  - How to apply operations to an object

Example:

```
String s;
s = new String (“hello”);
StdOut.pringln(s.substring(0,3));
```

You have already done this many times

Use new to invoke a constructor, use the data type name to specify the type of object, to apply operations invoke a method by using the object name and the “.”, dot, operator.

Abstract Data Type

- A data type is a set of values and a set of operations on those values
- In an abstract data type the representation is hidden from the client
- Besides Strings there are many very useful ADTs including stacks, lists, queues
- You will also make ADTs yourself

Color ADT

Color is a sensation in the eye from electromagnetic radiation.

An ADT allows us to write Java programs that manipulate color.

Values

		examples							
R (8 bits)	red intensity	255	0	0	0	255	0	119	105
G (8 bits)	green intensity	0	255	0	0	255	64	33	105
B (8 bits)	blue intensity	0	0	255	0	255	128	27	105
color									

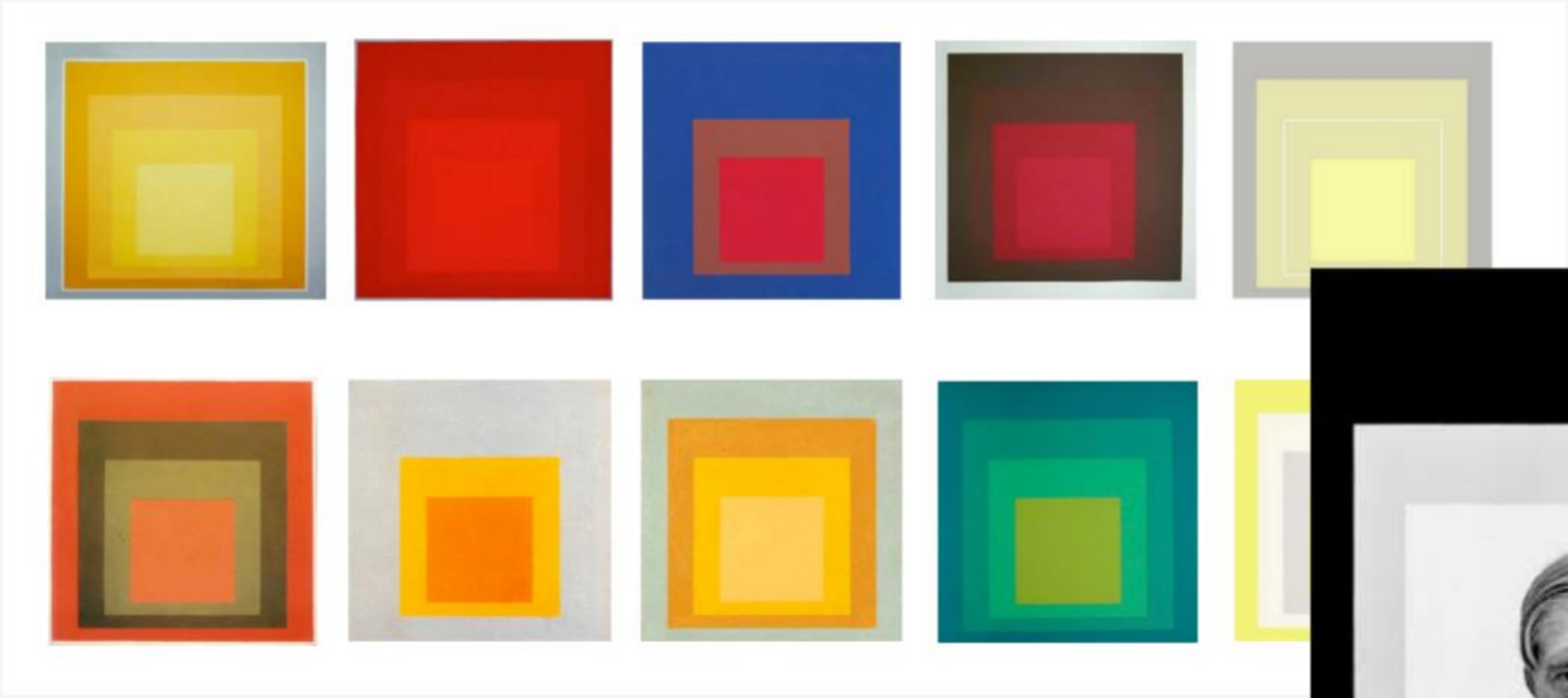
API (operations)

public class Color	
Color(int r, int g, int b)	
int getRed()	*red intensity*
int getGreen()	*green intensity*
int getBlue()	*blue intensity*
Color brighter()	*brighter version of this color*
Color darker()	*darker version of this color*
String toString()	*string representation of this color*
boolean equals(Color c)	*is this color the same as c's ?*

If a color is very close, adjacent to the color of the background, it almost gets absorbed by the background, like a silent dialogue, while contrasting colors create a loud expression.

Albers squares

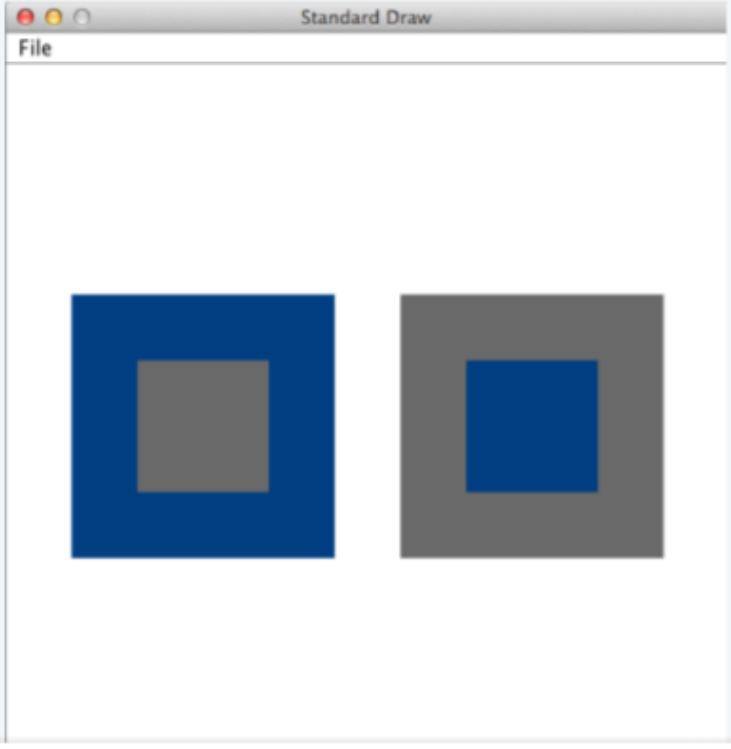
Josef Albers. A 20th century artist who revolutionized the way people think about color.



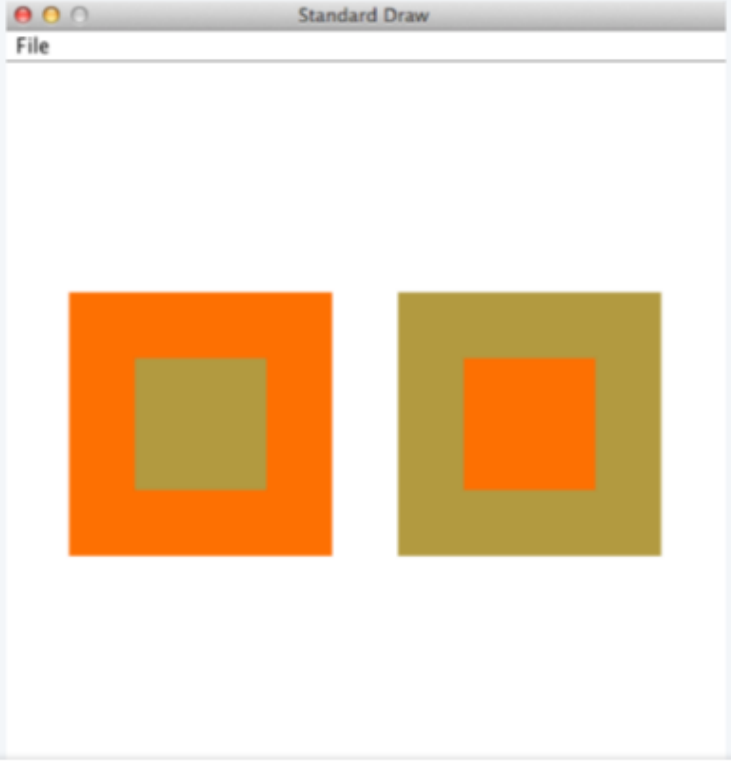
Josef Albers 1888–1976

Color client example: Albers squares

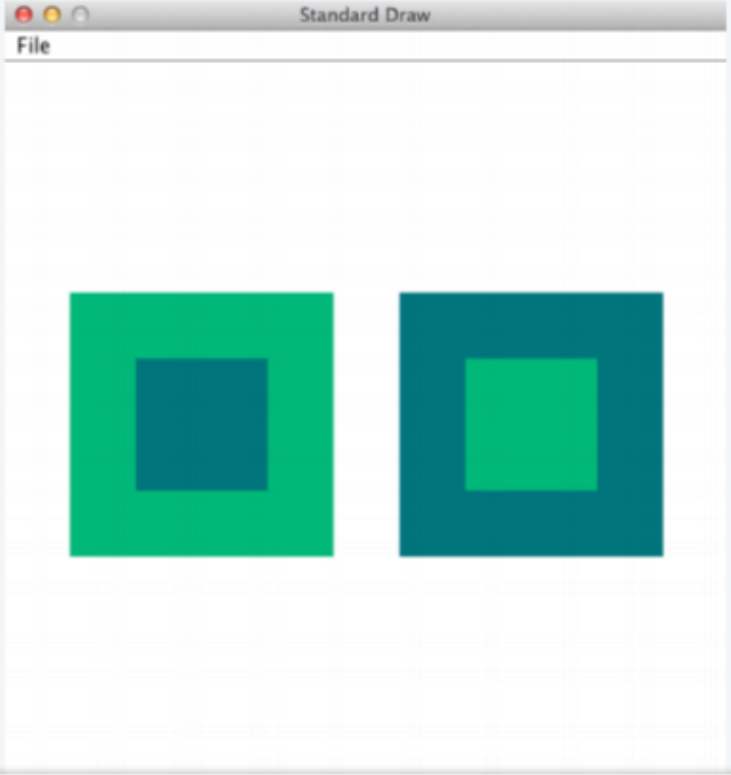
Goal. Write a Java program to generate Albers




```
% java AlbersSquares 0 64 128 105 105 105
```




```
% java AlbersSquares 251 112 34 177 153 71
```




```
% java AlbersSquares 28 183 122 15 117 123
```

Color client example: Albers squares

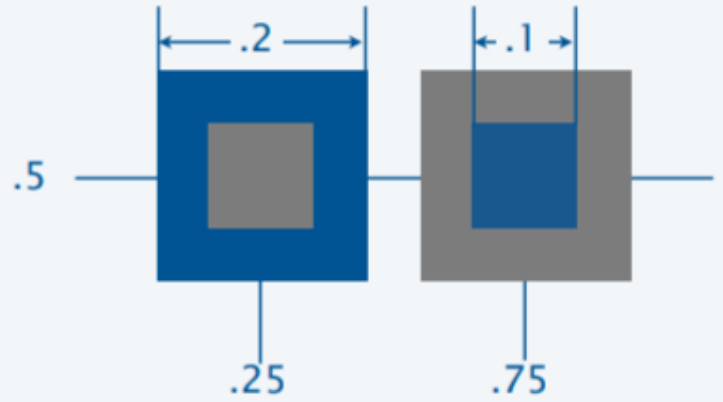
```
public class AlbersSquares
{
   public static void main(String[] args)
   {
      int r1 = Integer.parseInt(args[0]);
      int g1 = Integer.parseInt(args[1]);
      int b1 = Integer.parseInt(args[2]);
      Color c1 = new Color(r1, g1, b1);

      int r2 = Integer.parseInt(args[3]);
      int g2 = Integer.parseInt(args[4]);
      int b2 = Integer.parseInt(args[5]);
      Color c2 = new Color(r2, g2, b2);

      StdDraw.setPenColor(c1);
      StdDraw.filledSquare(.25, .5, .2);
      StdDraw.setPenColor(c2);
      StdDraw.filledSquare(.25, .5, .1);

      StdDraw.setPenColor(c2);
      StdDraw.filledSquare(.75, .5, .2);
      StdDraw.setPenColor(c1);
      StdDraw.filledSquare(.75, .5, .1);
   }
}
```

← create first color

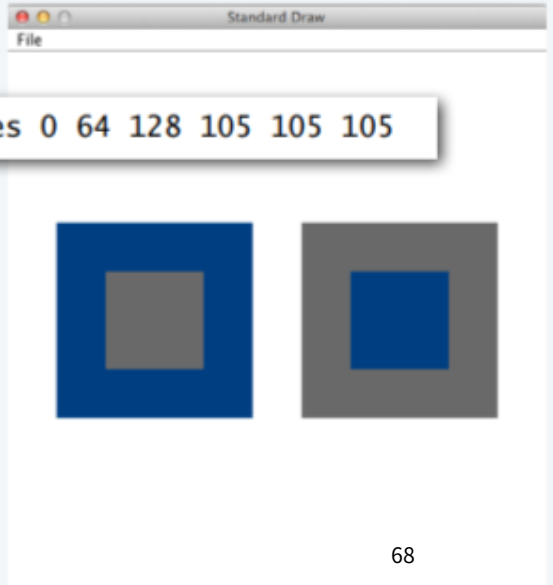
← create second color

← draw first square

← draw second square




```
% java AlbersSquares 0 64 128 105 105 105
```



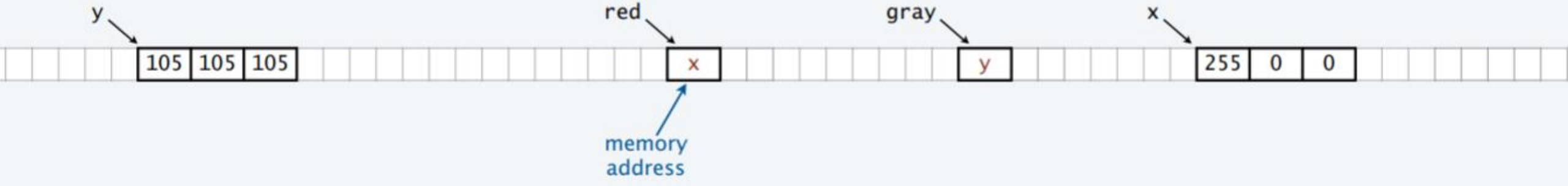
https://introcs.cs.princeton.edu/java/stdlib/

OOP context for color

Q. How does Java represent color? Three `int` values? Packed into one `int` value?

A. We don't know. The representation is hidden. It is an *abstract* data type.

Possible memory representation of `red = new Color(255, 0, 0)` and `gray = new Color(105, 105, 105);`



An object reference is analogous to a variable name.

- It is not the value but it refers to the value.
- We can manipulate the value in the object it refers to.
- We can pass it to (or return it from) a method.

We also use object references to *invoke* methods (with the . operator)

References and abstraction

René Magritte. This is not a pipe.

← It is a picture of a painting of a pipe.

Java. These are not colors.

```
public static Color toGray(Color c)
{
   int y = (int) Math.round(lum(c));
   Color gray = new Color(y, y, y);
   return gray;
}
```

Object-oriented programming. A natural vehicle for studying abstract models of the real world.

"This is not a pipe."

Yes it is! He's referring to the physical object he's holding.
Joke would be better if he were holding a *picture* of a pipe.

This is not a pipe.

Surrealist computer scientist:
Neither is this.

```
% java RandomSeq 10000 | java Average
```

Computing with color: monochrome luminance

Def. The *monochrome luminance* of a color quantifies its effective brightness.

NTSC standard formula for luminance: 0.299r + 0.587g + 0.114b.

```
import java.awt.Color;
public class Luminance
{
   public static double lum(Color c)
   {
      int r = c.getRed();
      int g = c.getGreen();
      int b = c.getBlue();
      return .299*r + .587*g + .114*b;
   }
   public static void main(String[] args)
   {
      int r = Integer.parseInt(args[0]);
      int g = Integer.parseInt(args[1]);
      int b = Integer.parseInt(args[2]);
      Color c = new Color(r, g, b);
      StdOut.println(Math.round(lum(c)));
   }
}
```

```
% java Luminance 0 64 128
52
```

	examples							
red intensity	255	0	0	0	255	0	119	105
green intensity	0	255	0	0	255	64	33	105
blue intensity	0	0	255	0	255	128	27	105
color								
luminance	76	150	29	0	255	52	58	105

Applications (next)

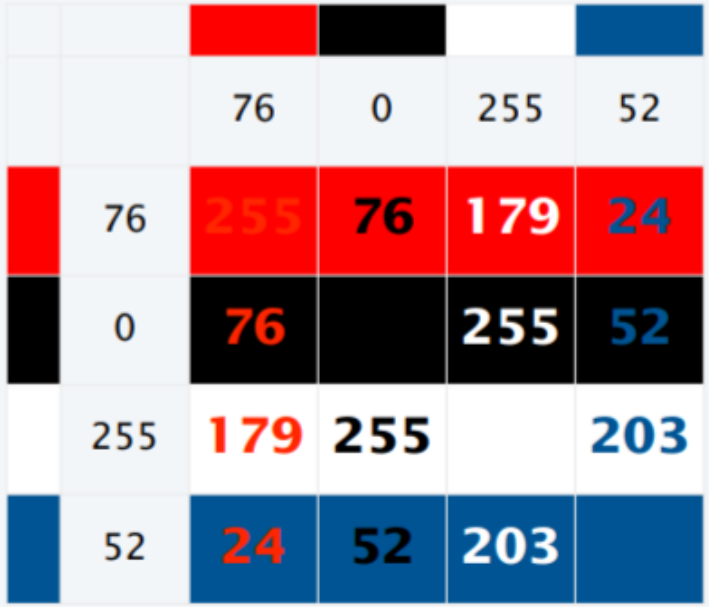
- Choose colors for displayed text.
- Convert colors to grayscale.

Computing with color: compatibility

Q. Which font colors will be most readable with which background colors on a display?

Rule of thumb. Absolute value of difference in luminosity should be > 128.

```
public static boolean compatible(Color a, Color b)
{
  return Math.abs(lum(a) - lum(b)) > 128.0;
}
```



Computing with color: grayscale

Goal. Convert colors to grayscale values.

Fact. When all three R, G, and B values are the same, resulting color is on grayscale from 0 (black) to 255 (white).

Q. What value for a given color?

A. Its luminance!

```
public static Color toGray(Color c)
{
   int y = (int) Math.round(lum(c));
   Color gray = new Color(y, y, y);
   return gray;
}
```

method for Luminance library

	examples							
red intensity	255	0	0	0	255	0	119	105
green intensity	0	255	0	0	255	64	33	105
blue intensity	0	0	255	0	255	128	27	105
color								
luminance	76	150	29	0	255	52	58	105
grayscale								

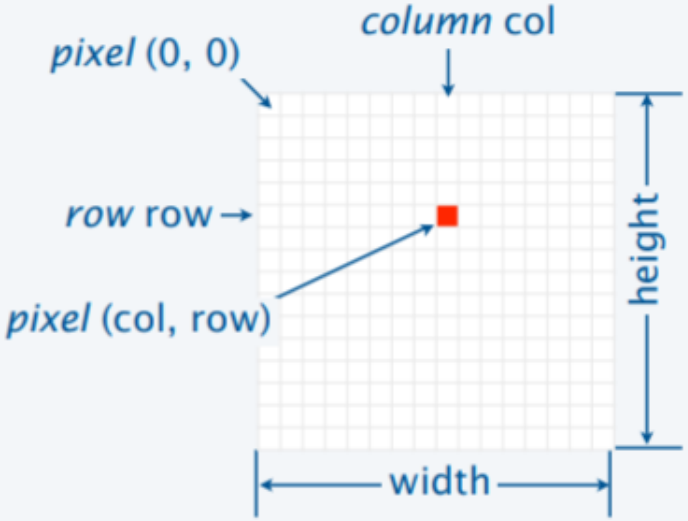
Picture ADT

A Picture is a 2D array of pixels.

defined in terms of its ADT values (typical)

An ADT allows us to write Java programs that manipulate pictures.

Values (2D arrays of Colors)

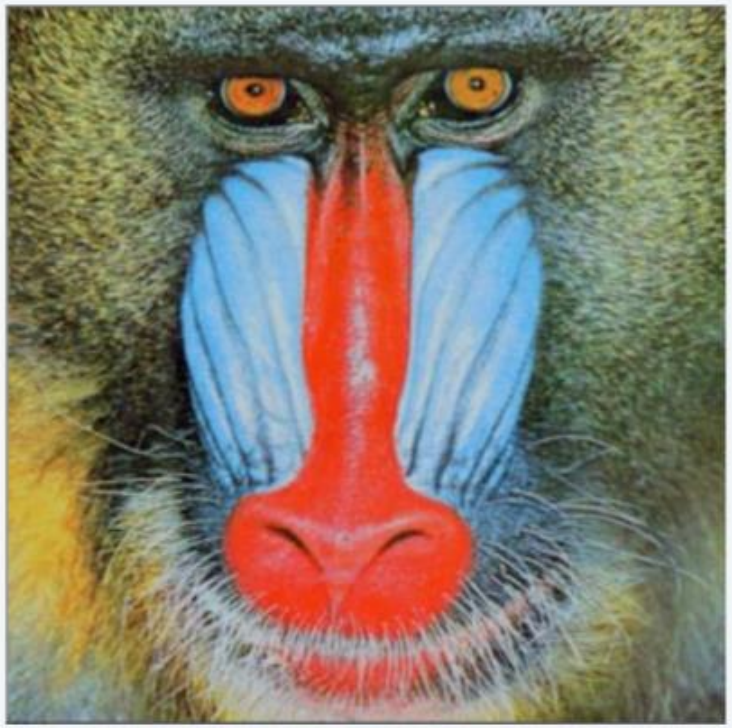


API (operations)

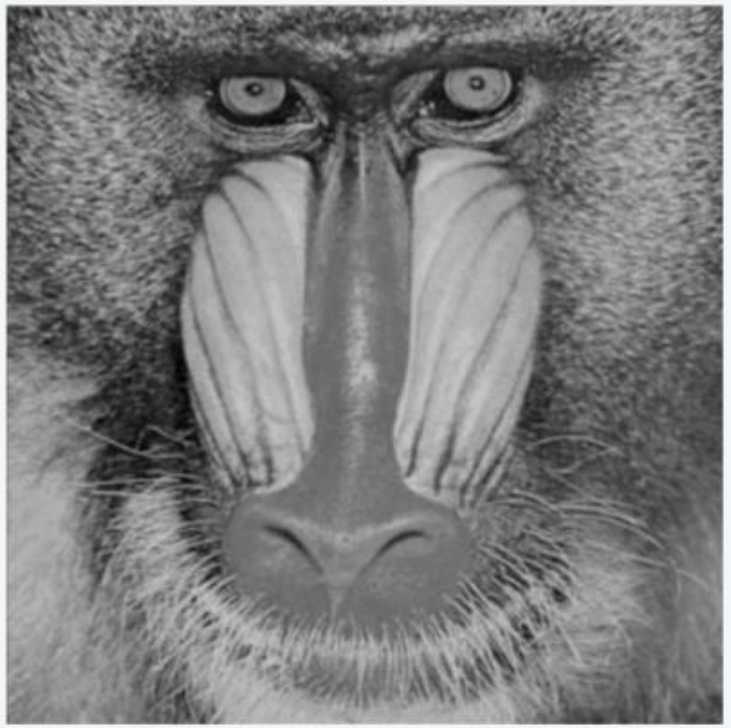
public class Picture	
Picture(String filename)	*create a picture from a file*
Picture(int w, int h)	*create a blank w-by-h picture*
int width()	*width of the picture*
int height()	*height of the picture*
Color get(int col, int row)	*the color of pixel (col, row)*
void set(int col, int row, Color c)	*set the color of pixel (col, row) to c*
void show()	*display the image in a window*
void save(String filename)	*save the picture to a file*

Picture client example: Grayscale filter

Goal. Write a Java program to convert an image to grayscale.



Source: mandrill.jpg



```
% java Grayscale mandrill.jpg
```

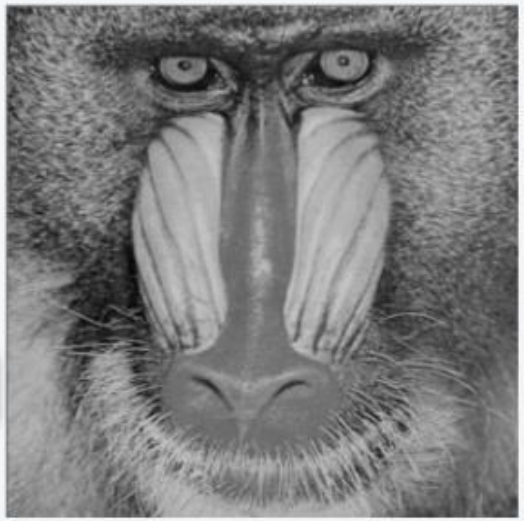
Picture client example: Grayscale filter

```
import java.awt.Color;
public class Grayscale
{
   public static void main(String[] args)
   {
      Picture pic = new Picture(args[0]);
      for (int col = 0; col < pic.width(); col++)
         for (int row = 0; row < pic.height(); row++)
         {
            Color color = pic.get(col, row);
            Color gray  = Luminance.toGray(color);
            pic.set(col, row, gray);
         }
      pic.show();
   }
}
```

create a new picture

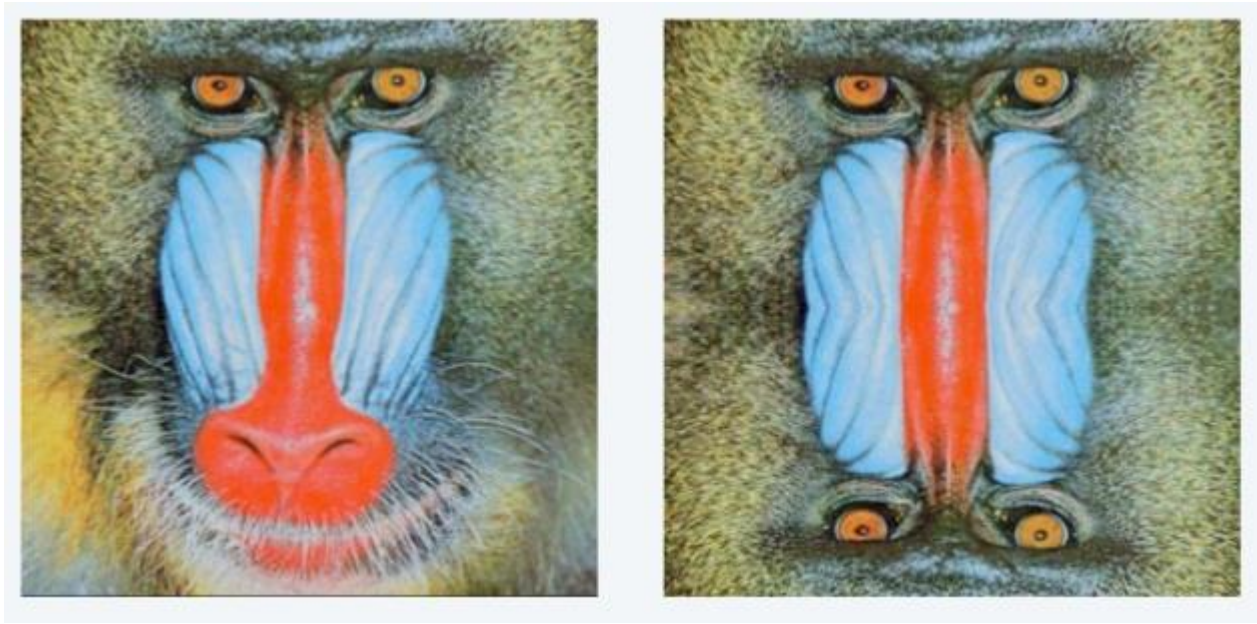
fill in each pixel

```
% java Grayscale mandrill.jpg
```



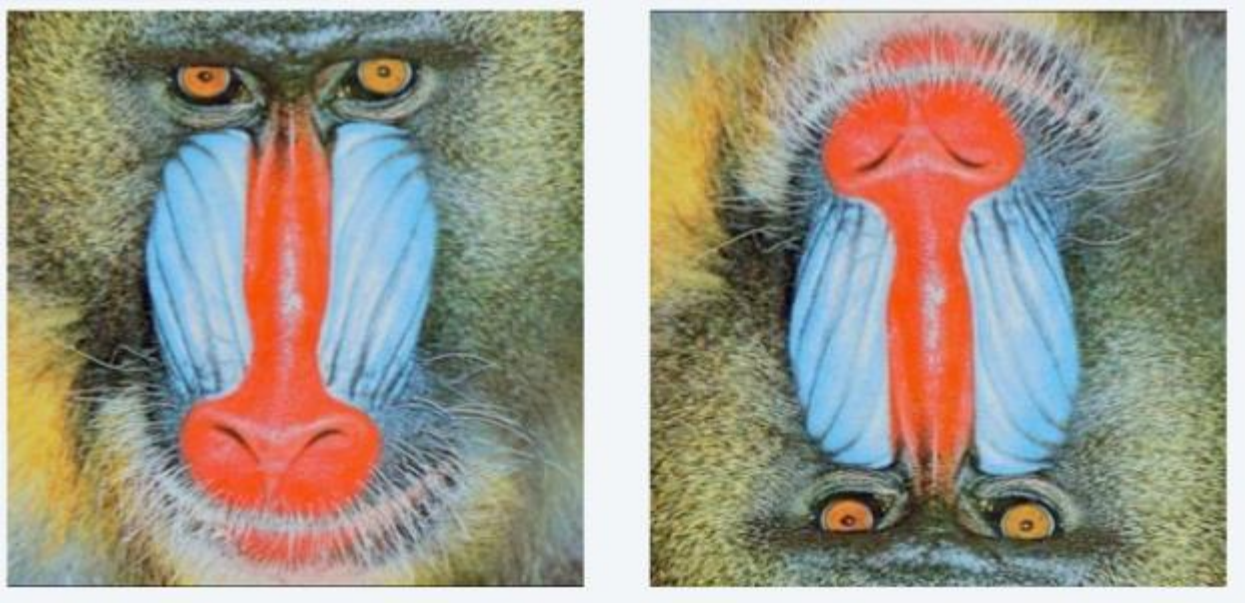
Flip a picture

```
Picture pic = new Picture(args[0]);
for (int col = 0; col < pic.width(); col++)
   for (int row = 0; row < pic.height(); row++)
      pic.set(col, pic.height()-row-1, pic.get(col, row));
pic.show();
```



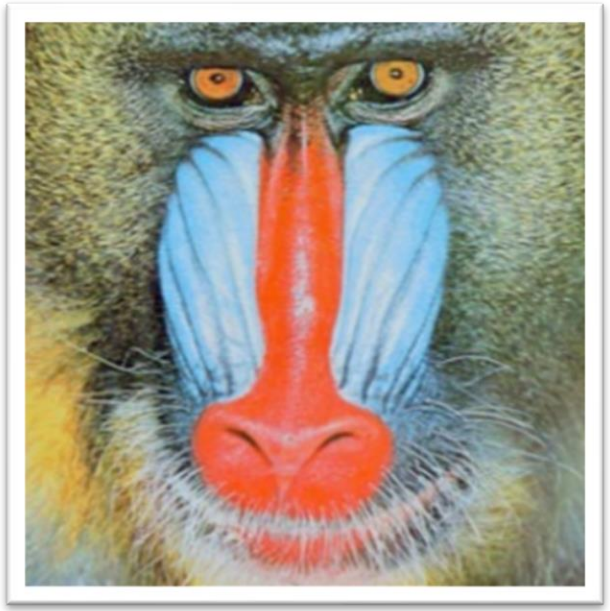
Flip a picture

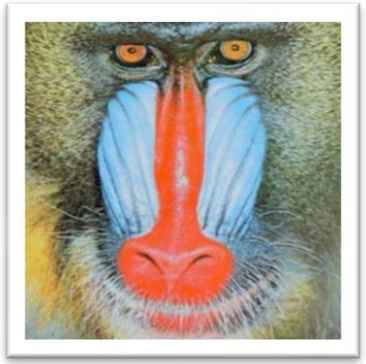
```
Picture source = new Picture(args[0]);
int width  = source.width();
int height = source.height();
Picture target = new Picture(width, height);
for (int col = 0; col < width; col++)
   for (int row = 0; row < height; row++)
      target.set(col, height-row-1, source.get(col, row));
target.show();
```

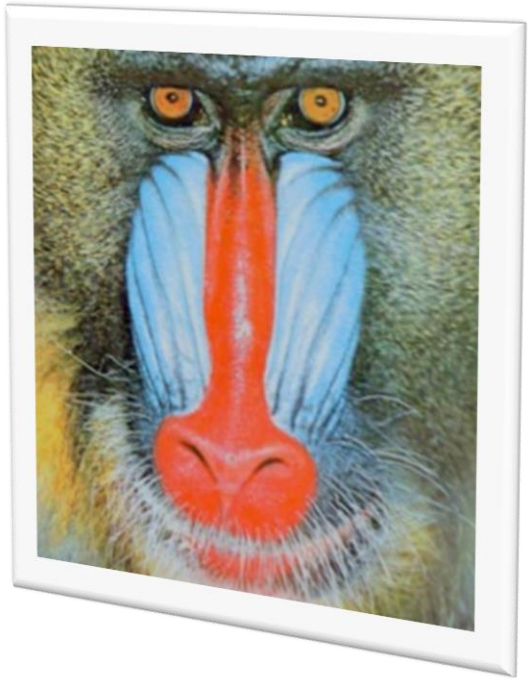


Processing each pixel in a picture is a common operation in computer graphics…

```
Picture pic = new Picture(args[0]);
for (int col = 0; col < pic.width(); col++)
   for (int row = 0; row < pic.height(); row++)
      pic.set(col, row, pic.get(col, row));
pic.show();
```



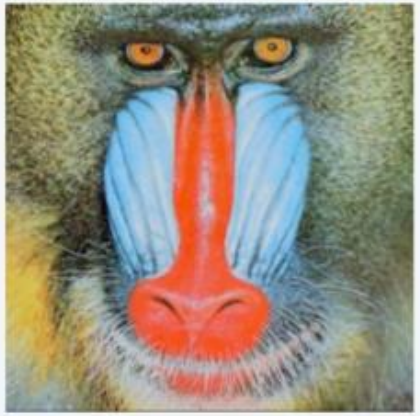




Picture client example: Scaling filter

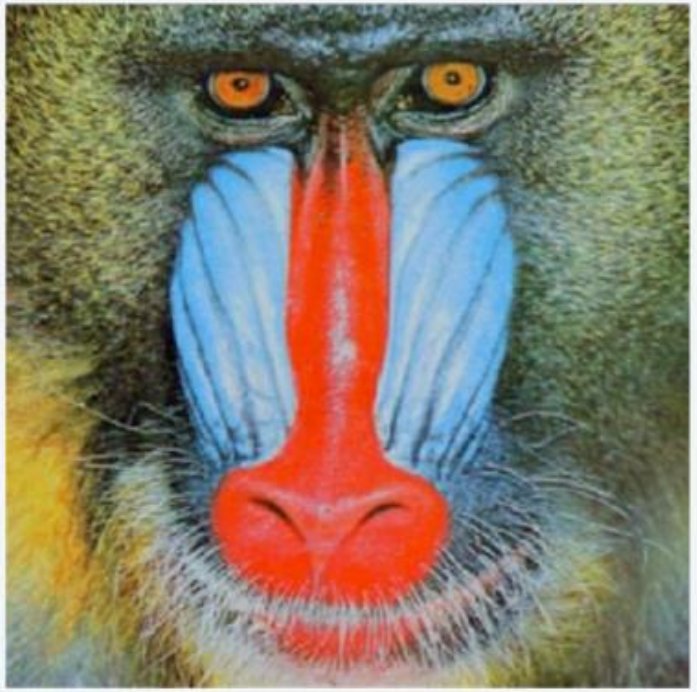
Goal. Write a Java program to scale an image (arbitrarily and independently on *x* and *y*).

300x300



Source: mandrill.jpg

500x500



```
% java Scale mandrill.jpg 500 500
```

600x200

```
% java Scale mandrill.jpg 600 200
```

99x99

```
% java Scale mandrill.jpg 99 99
```

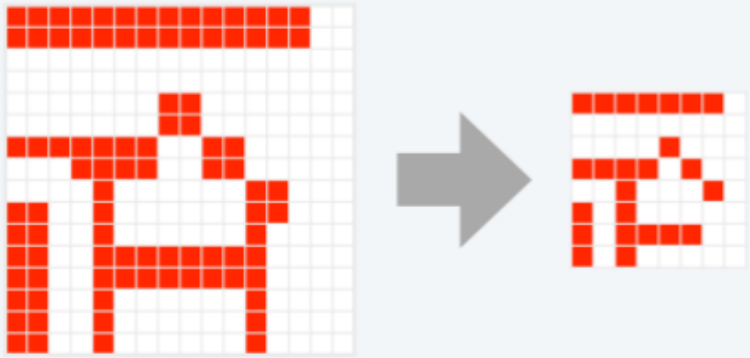
200x400

```
% java Scale mandrill.jpg 200 400
```

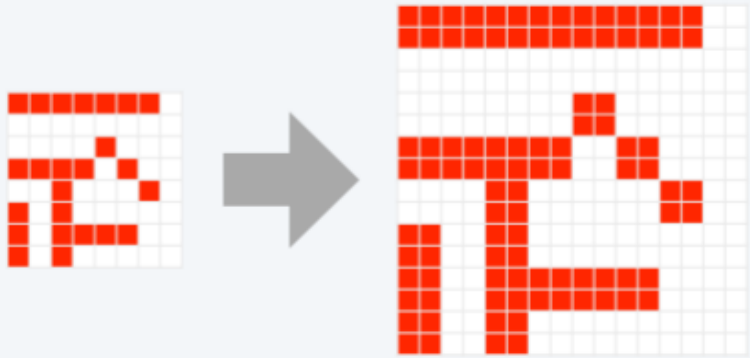
Picture client example: Scaling filter

Goal. Write a Java program to scale an image (arbitrarily and independently on *x* and *y*).

Ex. Downscaling by halving.
Shrink in half by deleting alternate rows and columns.



Ex. Upscaling by doubling.
Double in size by replacing each pixel with four copies.



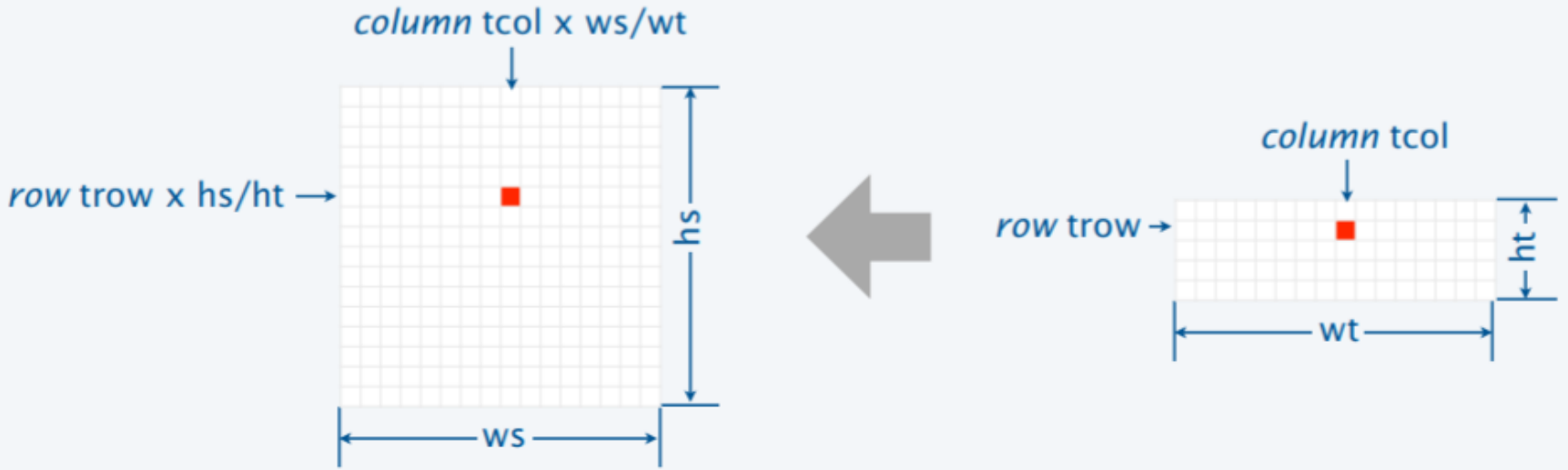
Picture client example: Scaling filter

Goal. Write a Java program to scale an image (arbitrarily and independently on *x* and *y*).

A uniform strategy to scale from ws-by-hs to wt-by-ht.

- Scale column index by ws/wt .
- Scale row index by hs/ht .

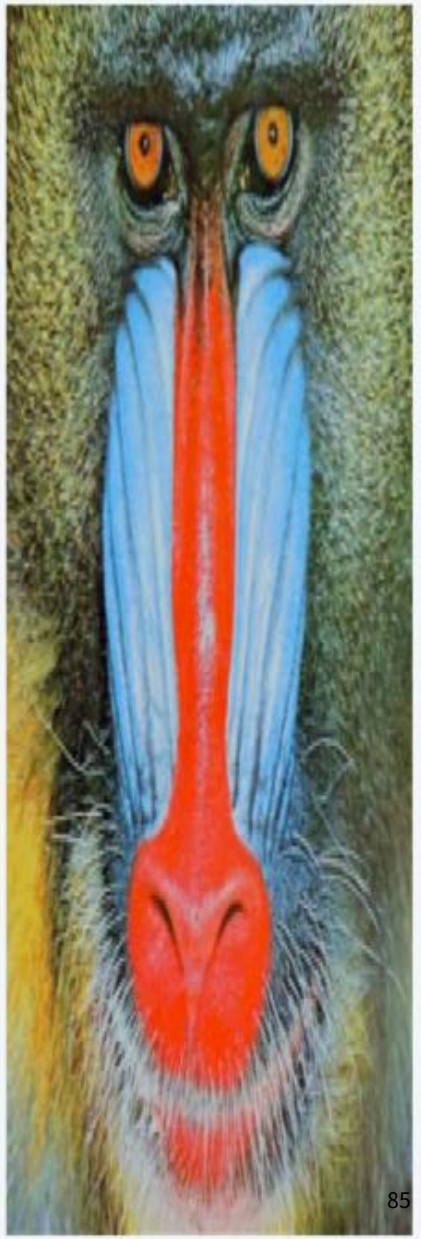
Approach. Arrange computation to compute exactly one value for each *target* pixel.



Picture client example: Scaling filter

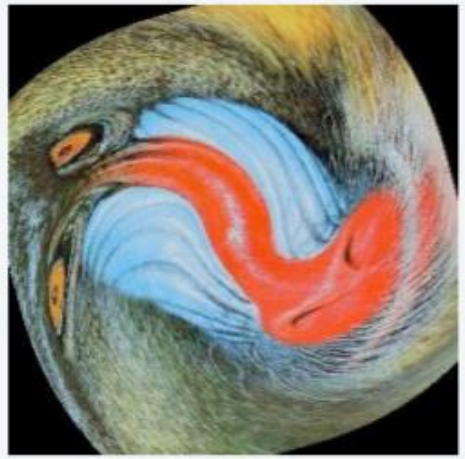
```
import java.awt.Color;
public class Scale
{
   public static void main(String[] args)
   {
      String filename = args[0];
      int w = Integer.parseInt(args[1]);
      int h = Integer.parseInt(args[2]);
      Picture source = new Picture(filename);
      Picture target = new Picture(w, h);
      for (int tcol = 0; tcol < w; tcol++)
         for (int trow = 0; trow < h; trow++)
         {
            int scol = tcol * source.width()  / w;
            int srow = trow * source.height() / h;
            Color color = source.get(scol, srow);
            target.set(tcol, trow, color);
         }
      target.show();
   }
}
```

```
% java Scale mandrill.jpg 300 900
```



More image-processing effects

RGB color separation



swirl filter

wave filter

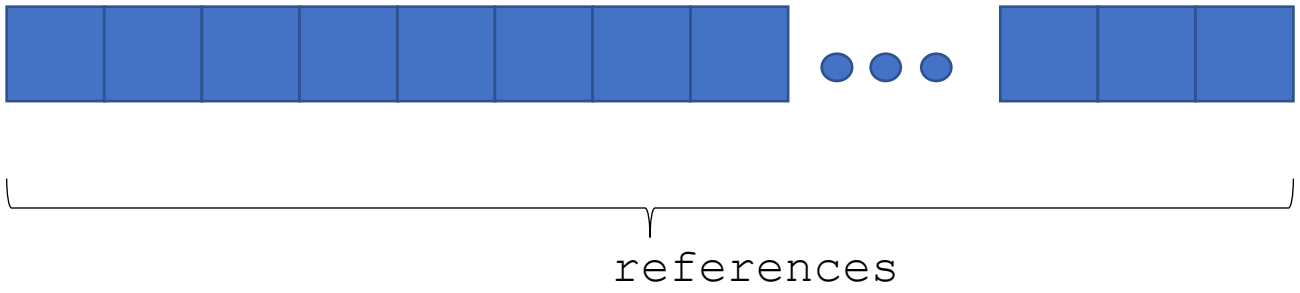
glass filter

Sobel edge detection

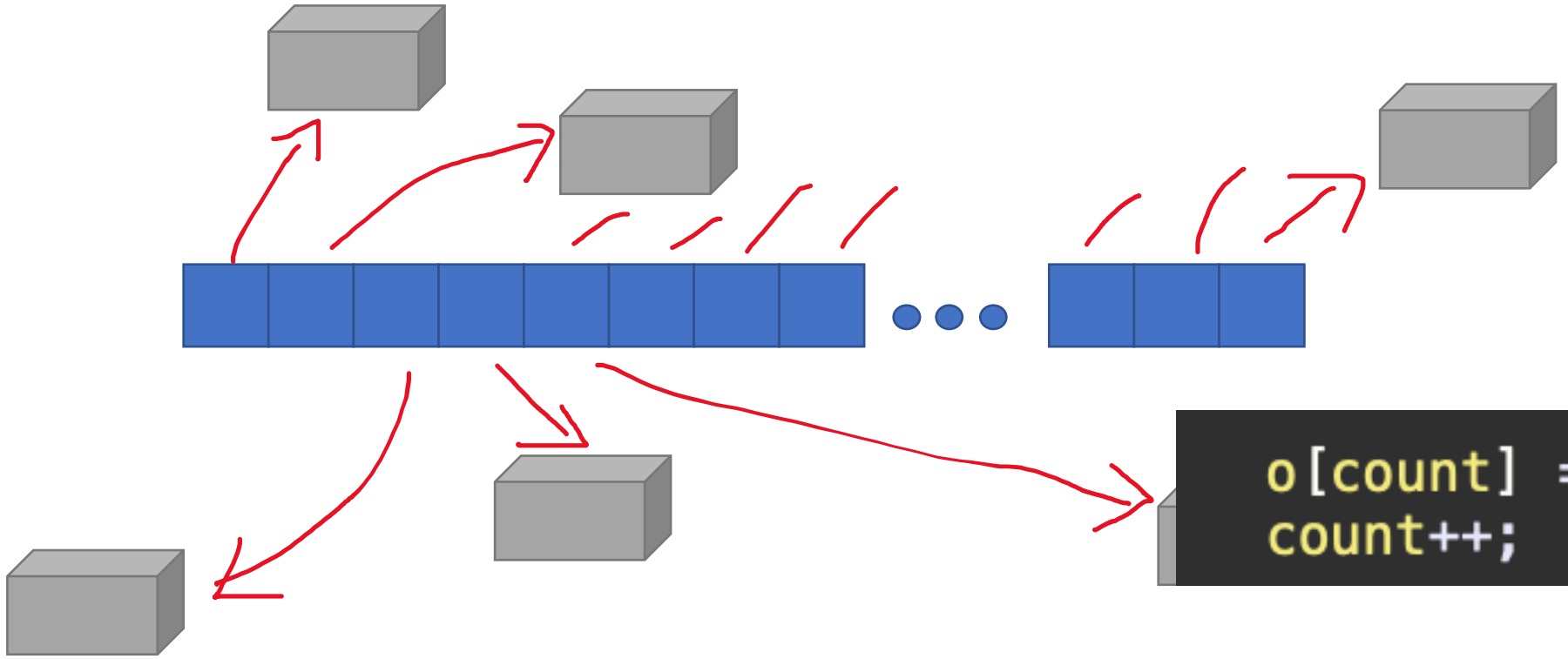
Working with Groups of Objects Using Arrays of Objects

When you create an array of objects, each element isn't an object it's just a place to store a reference. (note: this is unlike arrays of base types such as int or float etc).

```
int size = 20;
Object[] o = new Object[size];
```



```
Object[] o = new Object[size];
int count = 0;
```

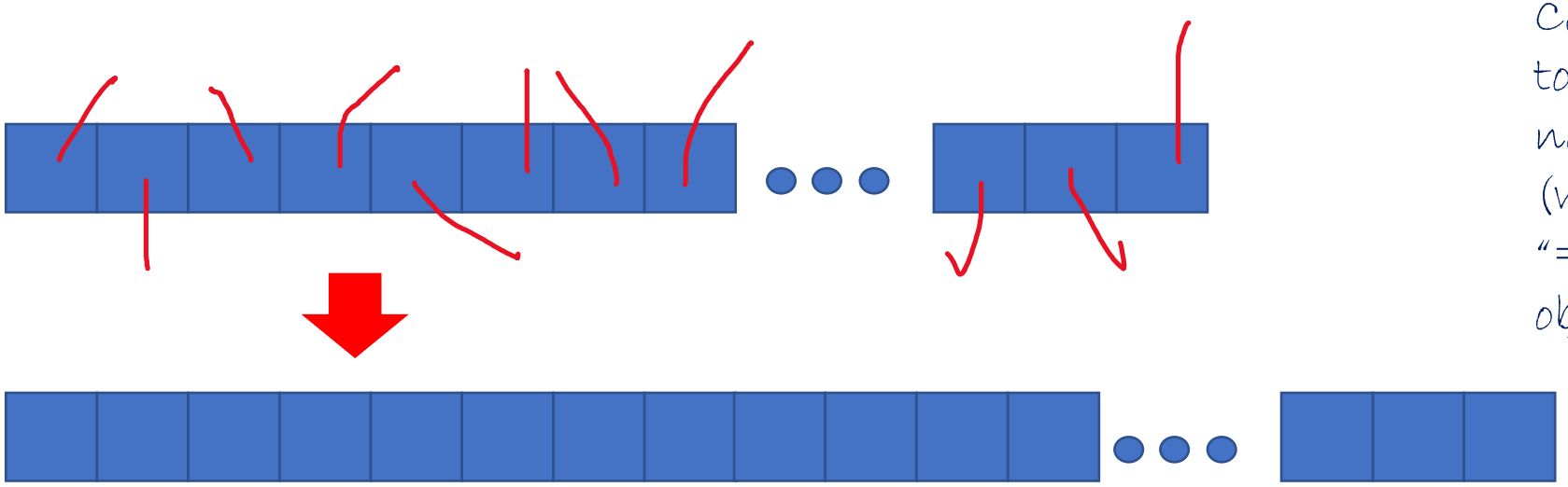


```
o[count] = new Object();
count++;
```

Each object is created one by one.

Each object added to the array must be separately created ("instantiated"), which is very different from arrays of base types… At first an object array is just an array of null pointers.

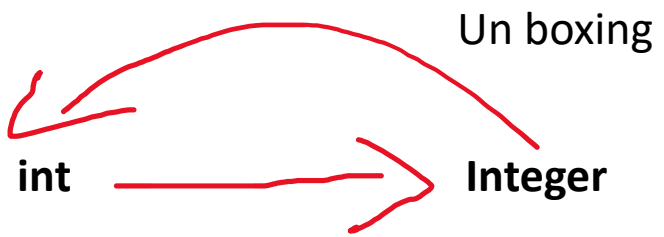
Because there is no limit on the memory required by objects using object references makes it easier to process arrays, for example if you exceed the size of an array you can create a bigger one just by copying the object references…



NB: Making a "Deep Copy" of an array means to the process of copying not just the references (which is the effect of "=") but of copying each object… in a loop

```
Object [] o2;
o2 = o;
o = null;
```

Boxing



“Boxing”, Unboxing, and Autoboxing

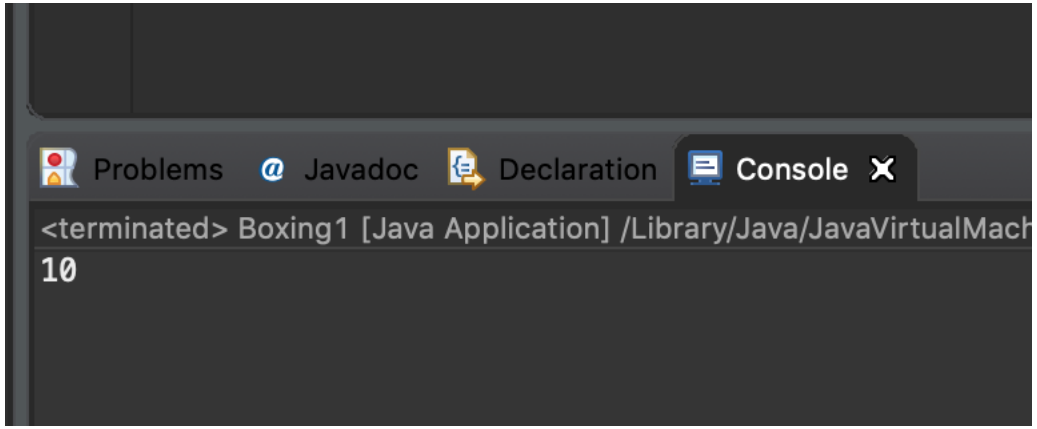
- ***Boxing*** is between the primitive types and their corresponding object wrapper classes.
- For example, converting an **int** to an **Integer**, a **double** to a **Double**
- If the conversion goes the other way, this is called ***unboxing***.
- ***Autoboxing*** is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.

- Here is the simplest example of autoboxing (a character a character object wrapper is assigned a character that is a primitive char type):
  - Character ch = 'a';

See also: https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html

If I'm calling a method that takes an Integer parameter with an int parameter, everything works fine.

```
2  public class Boxing1 {
3
4      public static void displayVal(Integer val) {
5          System.out.println(val);
6      }
7
8
9  public static void main(String[] args) {
10     int n = 10; displayVal(n);
11     }
12 }
```



Works OK

Compilation error

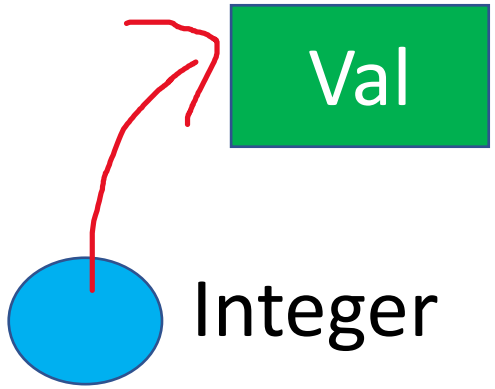
Disaster

- This one with an array fails and makes us sad.
- The reason for the problem is that int[] cannot be converted to Integer[] .
- **Boxing/unboxing know how to convert between base type and object, but not between object and object**
- An array is an object, it's created by calling new, when the array is created we "lost" the base type.

```
2  public class Boxing2 {
3
4
5      public static void displayVal(Integer[] valarr) {
6
7          for (int i = 0; i < valarr.length; i++) {
8              System.out.println(valarr[i]); }
9      }
10
11     public static void main(String [] args) {
12
13         int [] n = new int[3];
14
15         for (int i=0; i < 3; i++) {
16             n[i] = i;
17         }
18
19         displayVal(n);
20     }
```

```
EXCEPTION IN THREAD "MAIN"
JAVA.LANG.ERROR:
UNRESOLVED COMPILATION
PROBLEM:
THE METHOD
DISPLAYVAL(INTEGER[]) IN
THE TYPE BOXING2 IS NOT
APPLICABLE FOR THE
ARGUMENTS (INT[])
```

int



Blue indicates a reference.

The relationship between int and Integer is straightforward.

Incorrect handling of Object references is a common source of bugs

```
public static int search(String key, String[] a)
{
        for (int i=0; i < a.length; i++){
                if (a[i] == key) return i;
        }
        return -1;
}
```

search(“Chloe”, a)

i	a[i]
1	Amelia
2	Olivia
3	Charlotte
4	Isla
5	Mia
6	Ava
7	Chloe
8	Grace
9	Sophia
10	Zoe

This fails because the == checks for equality of the reference at a[i]

```
public static int search(String key, String[] a)
{
        for (int i=0; i < a.length; i++){
                if (a[i].compareTo(key) == 0)
                        return i;
        }
        return -1;
}
```

search(“Chloe”, a)

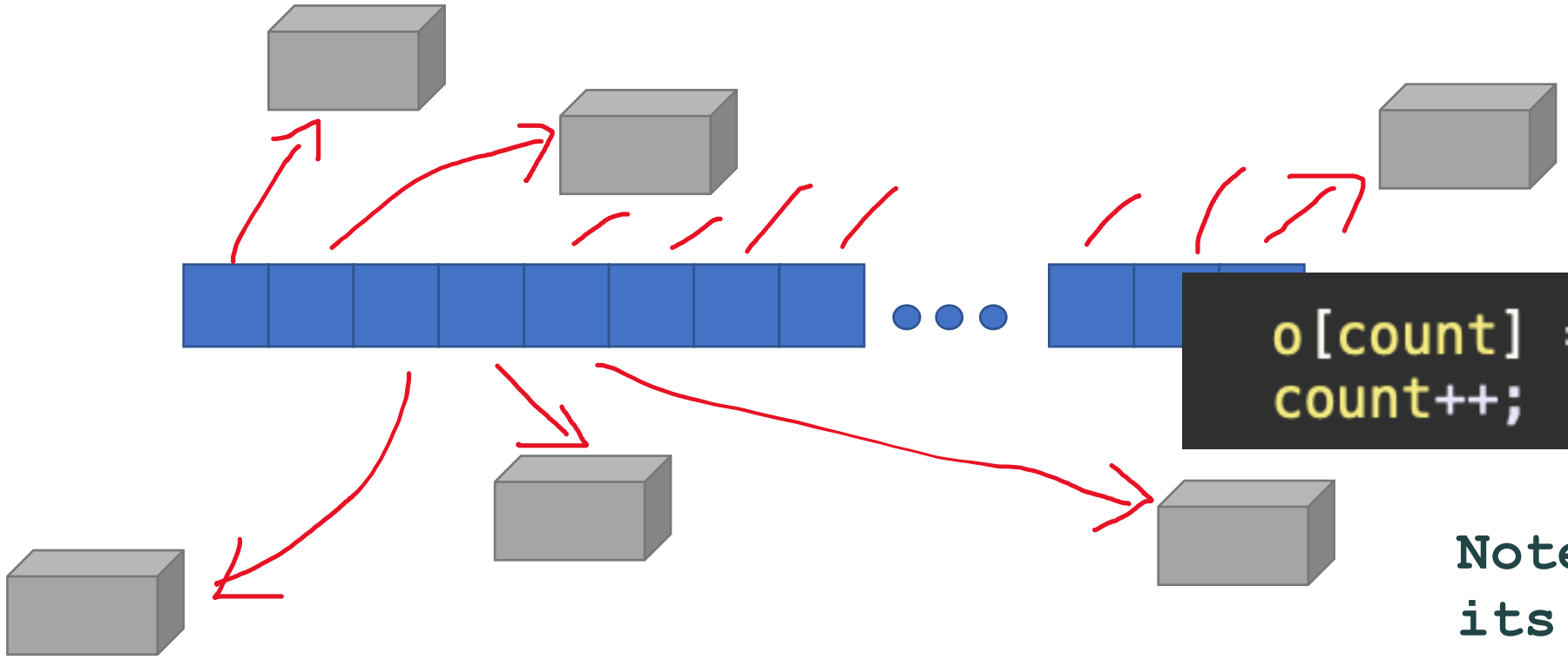
i	a[i]
1	Amelia
2	Olivia
3	Charlotte
4	Isla
5	Mia
6	Ava
7	Chloe
8	Grace
9	Sophia
10	Zoe

Fixed by using the . Operator to call the object compareTo method.

Introduction to Collections using Generic Types

Searching in an array of Objects

```
Object[] o = new Object[size];
int count = 0;
```



```
o[count] = new Object();
count++;
```

The simplest group of objects is just an array of objects.

Note: initially its just an array of empty references

Searching Collections of Objects

- Common operations that involve searching through groups of objects

- **contains** (Object obj)
  - True if the group contains the object
- **indexOf**(Object obj)
  - Returns the index of the first occurrence of the object
  - See also lastIndexOf
- **remove**(Object obj)
  - Removes object from the collection

List ADT

- A list can be implemented in different ways, we will later see how using an array, or with pointers, or even a tree.
- However the client of the list requires that the list is able to provide certain basic operations including these.
- We consider a list datatype to be a set of possible lists and a set of operations that can be performed (says nothing about how the list is implemented).
- **An array of objects is not itself a list data type because it does not specify the operations that can be performed – when we specify operations and types a list can hold and write a class we can make a list data type.**

```java
public class City {
private String name;
private String country;
private int population;

public City(String n, String c, int p) {
    name = n;
    country = c;
    population = p;
    }


    public String toString() {
        return name + "(" + country + ") - "
                + Integer.toString(population);

    }


}
```

Suppose that we want to make an array of City objects and call it cities...

Create an array

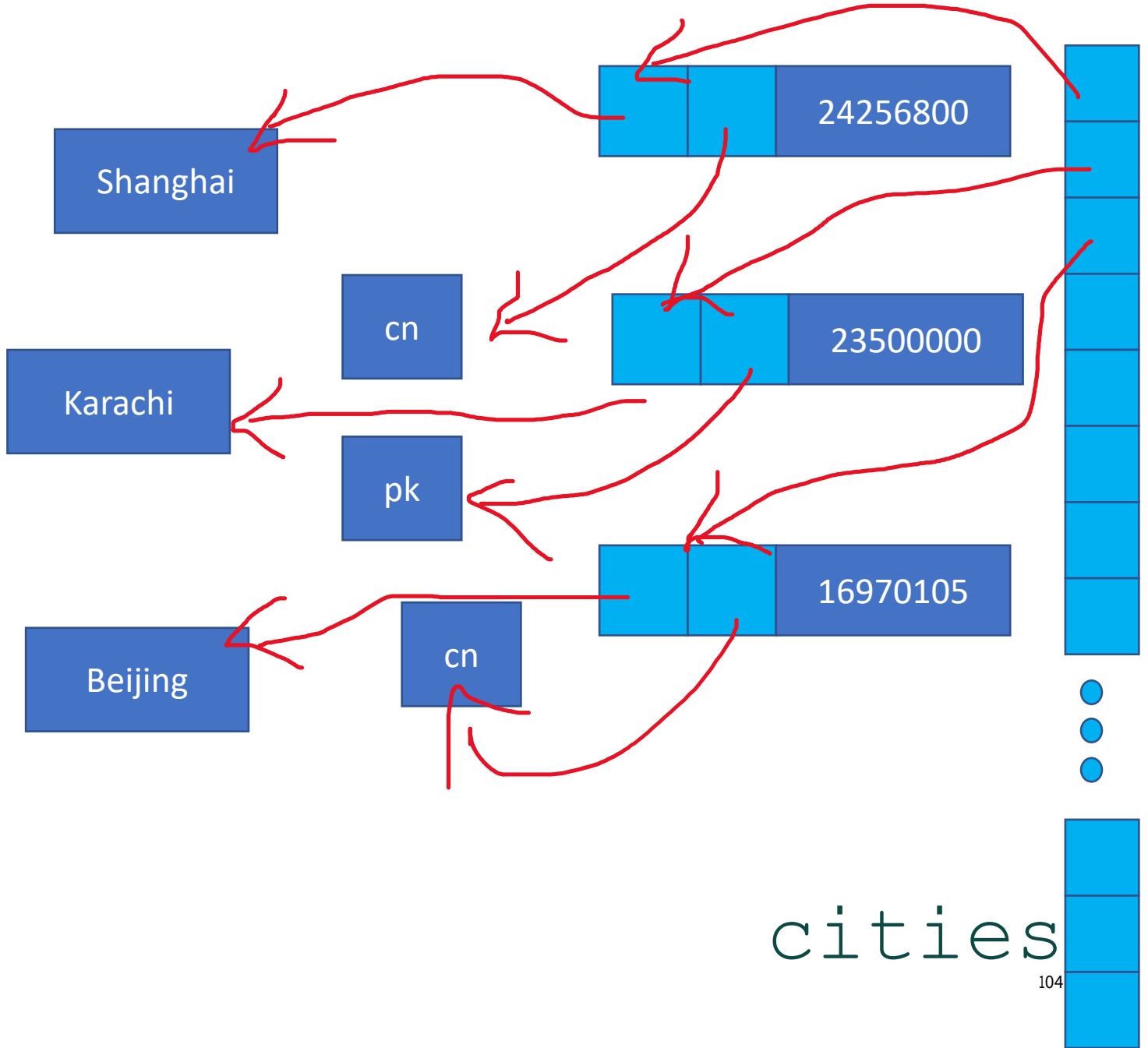
```
#name, country, population
Shanghai,cn,24256800
Karachi,pk,23500000
Beijing,cn,21516000
Dhaka,bd,16970105
Delhi,in,16787941
Lagos,ng,16060303
Istanbul,tr,14025000
Tokyo,jp,13513734
Guangzhou,cn,13080500
…
```

```
public static void main(String args[]) {
    City[] cities = new City[200];
```

Open the file to read it into the array

It's very common to load data in memory from external datafiles or from databases.

When we load everything in memory, every object (including Strings) becomes a reference to the actual values. To search the array, we'll just start at index 0 in the array and increase the index.



```
private String country;
private int population;

public City(String n, String c, int p) {
    name = n;
    country = c;
    population = p;
    }


    public boolean isNamed(String name) {
        return this.name.equals(name);
    }
```

```
    int i = 0;
    String name = args[0];
    while (!cities[i].isNamed(name)) {
        i++;

    }
```

Searching to see which city is named some particular string

We assumed that each name occurs once – means it is unique!

Time is proportional to the number of cities.

O(n)

Bigger array = slower

We can do better…

BINARY SEARCH

We can do far better with an array by running a binary search, which is the kind of search you run when looking for a word in a dictionary: you open in the middle, check a word there, and search either the first half or the second half.

You wouldn't be able to search a dictionary (otherwise than reading every page) if words were not ordered into it. A binary search can only work if the array is sorted. Class Arrays implements static methods that do that efficiently.

Precondition: the array must be sorted.

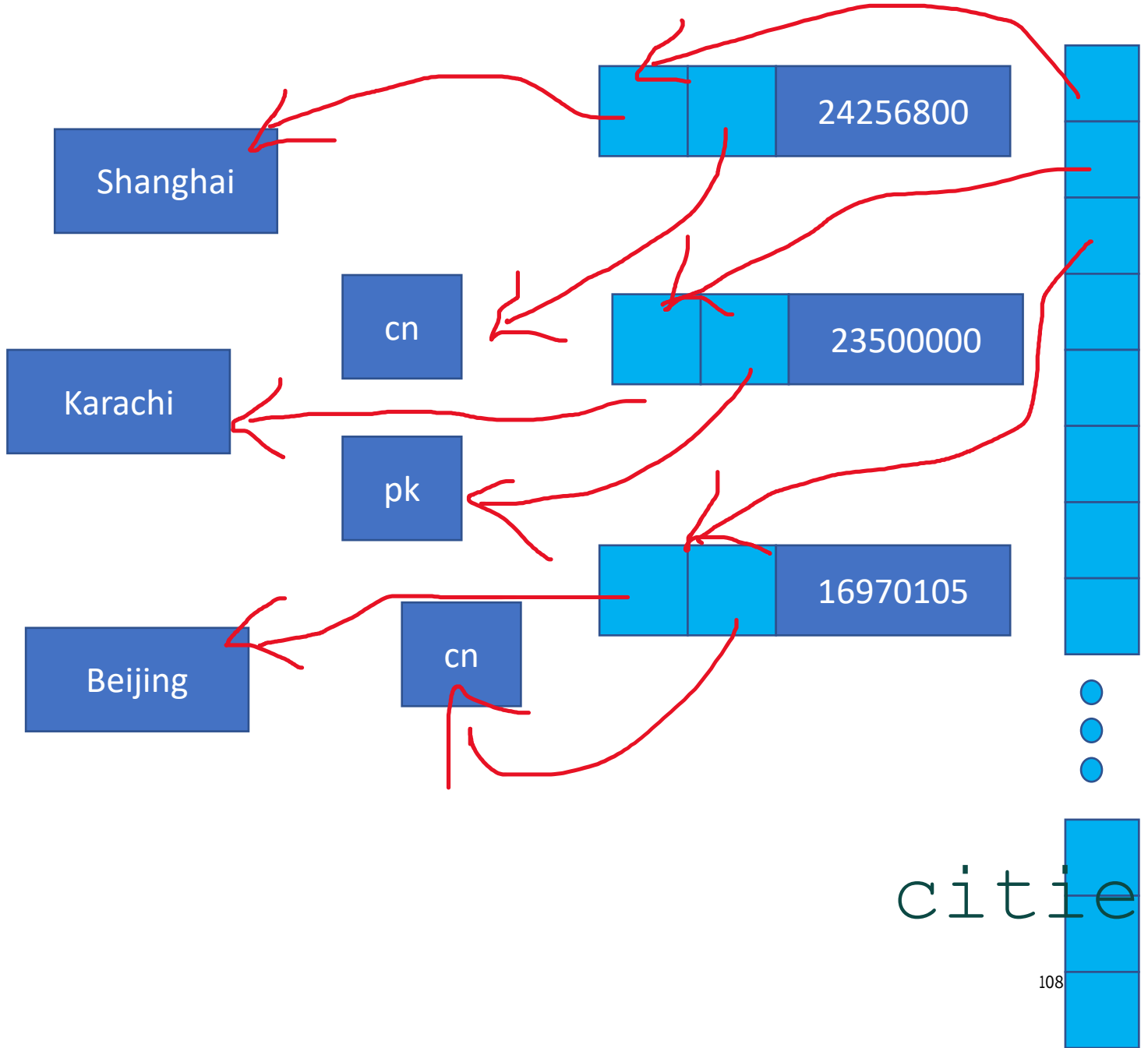
```
import java.util.Arrays;
Arrays.sort(cities, 0, cityCount);
```

from

to

Except objects can be complicated. What does "bigger" mean here?

- Population?
- Lexographic order of names?
- Lexographic order of countries?
- Something else?



- Arrays.sort() needs to know...
- In fact, it will require the existence of a method called compareTo() that it will call for organizing objects in the correct order.

You can’t sort if you can’t

COMPARE

java.lang

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

All Known Subinterfaces:

ChronoLocalDate, ChronoLocalDateTime<D>, Chronology, ChronoZonedDateTime
ScheduledFuture<V>

All Known Implementing Classes:

AbstractChronology, AbstractRegionPainter.PaintContext.CacheMode, Access
AddressingFeature.Responses, Authenticator.RequestorType, BigDecimal, Bi
CertPathValidatorException.BasicReason, Character, Character.UnicodeScri
ClientInfoStatus, CollationKey, Collector.Characteristics, Component.Bas
CRLReason, CryptoPrimitive, Date, Date, DayOfWeek, Desktop.Action, Diagn
Dialog.ModalityType, DocumentationTool.Location, Double, DoubleBuffer, D
File, FileTime, FileVisitOption, FileVisitResult, Float, FloatBuffer, Fo
FormSubmitEvent.MethodType, GraphicsDevice.WindowTranslucency, Gregorian
HijrahDate, HijrahEra, Instant, IntBuffer, Integer, IsoChronology, IsoEr

Reminder: Interface

When a specially named function has to exist in a class, it's said that the class must *implement* an interface.

Let's review what an interface is.

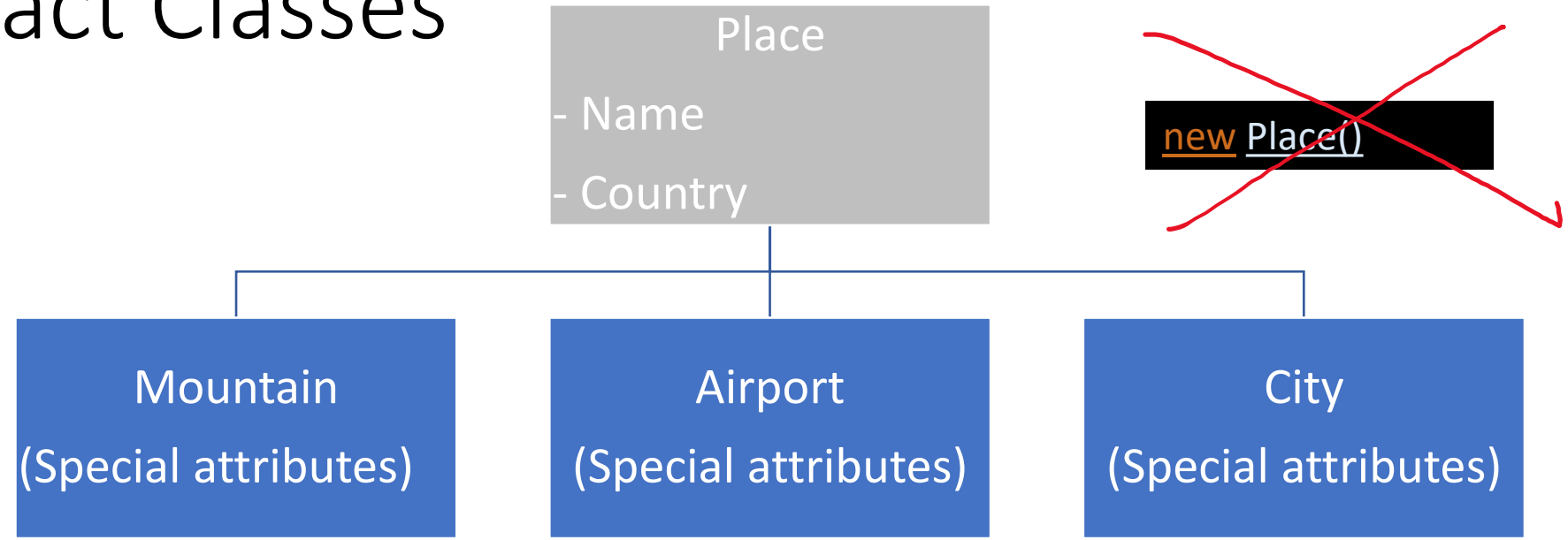
Interfaces – lightweight classes / templates

- No variable attributes allowed
- Constants are allowed
- Define methods that classes must implement to conform

- Abstract class = implicit interface definition

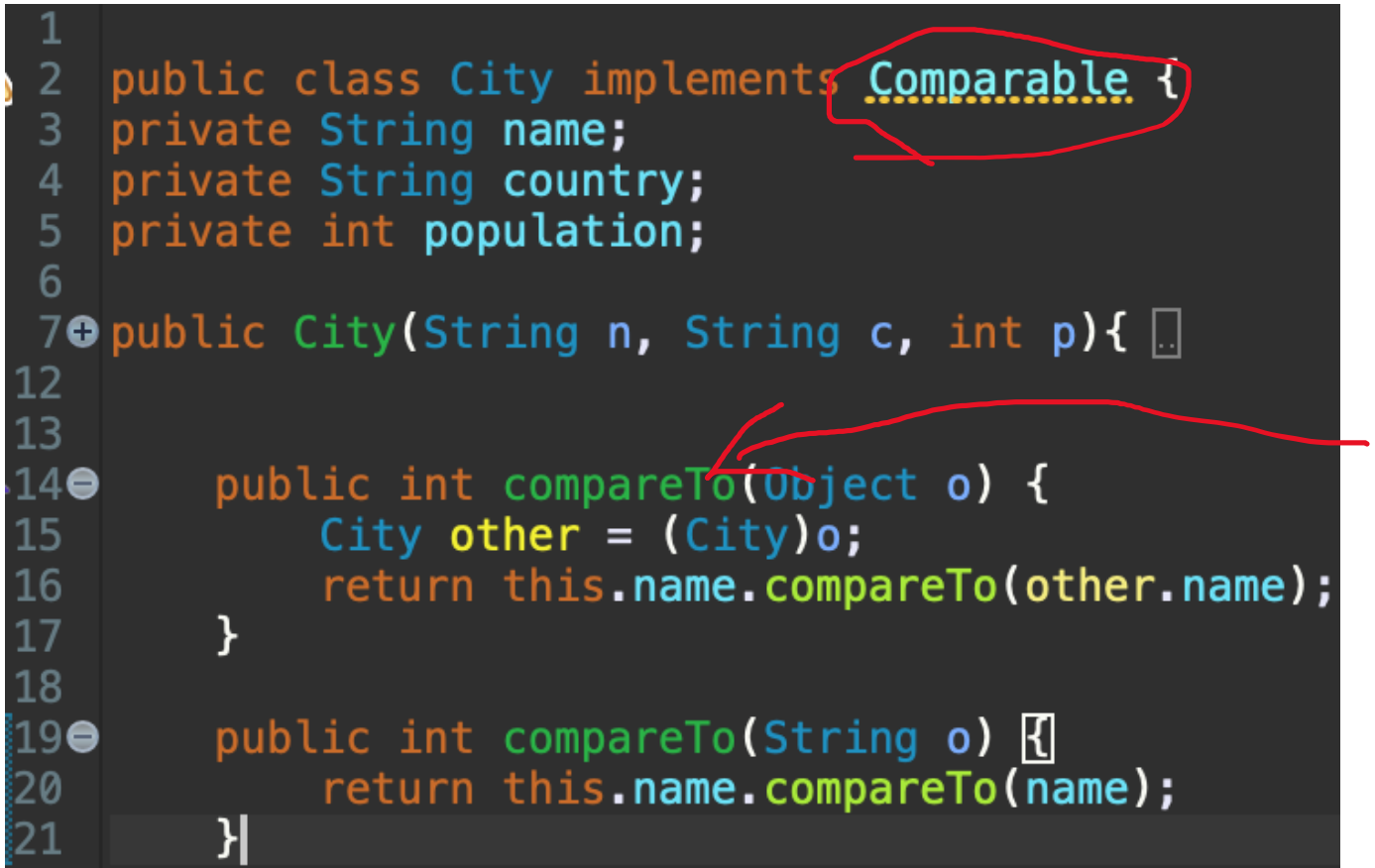
Interfaces are special lightweight classes that mostly define a behavior, through methods. If a class says that it implements an interface, then it must have the methods defined in the interface.

Abstract Classes



- A class can also be declared Abstract (means the opposite of ”real”)
- An abstract class cannot be directly used to instantiate an object – instead you must first create a new class that extends the abstract one
- An abstract class can have methods defined, but also indicate that it **should** have a method and force others who extend the class to write the method in the child classes

Arrays.sort() only works if the class implements the Comparable interface, which requires a compareTo() method. Here we say that comparing cities means comparing their names.



Need for the Comparable interface.

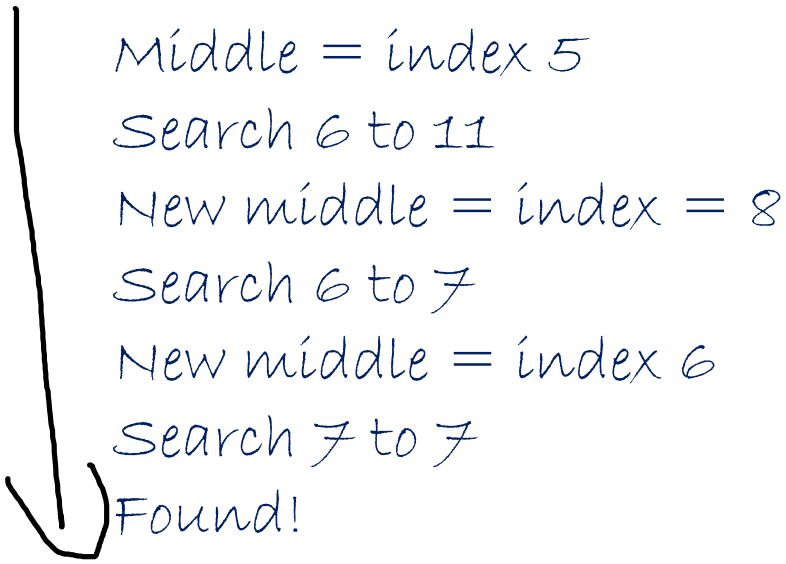
Required by Arrays.sort()

Custom string comparator we will use

0	Aberdeen
1	Boston
2	Chihuahua
3	Edinburgh
4	Istanbul
5	Liverppol
6	London
7	New York
8	Reykjavik
9	Rio De Janeiro
10	Shanghai
11	Tokyo

Find New York

12 Elements



Remember we saw that binary search works by reducing the size of the part of the array that is searched at each step by half

Number of comparisons in Binary search

- Size of the array is N
- Sequential search: N/2
  - If we double the number of items, we double the number of comparisons
- Binary Search: 2*log_2(N-1)
  - If we double the number of items we add one more comparison
  - It's a very efficient algorithm

$O(\log n)$

```java
public class Util {

    static City binarySearch(City[] arr,
                             int elements,
                             String lookedFor) {

        // assume the array is already sorted
        int start = 0;
        int end = elements - 1;
        int mid = 0;
        int comp;
        boolean found = false;

        while (start <= end) {
            mid = (start + end) / 2;
            comp = arr[mid].compareTo(lookedFor);
            if (comp < 0) {
                // array element is smaller
                start = mid + 1;
            } else if (comp > 0) {
                // array element is bigger
                end = mid - 1;
            } else {
                // found
                found = true;
                start = end + 1;
            }
        }
        if (found) {
            return arr[mid];
        } else {
            return null;
        }

    }
```

This is how it can be written in Java

Iteratively…

Or we can do it recursively

Trivial case 0 or 1

Although this is a case where recursion doesn't make the code all that much simpler

Class Arrays contains several (static) binarySearch() methods. You don't need to write it ...

In practice these classic algorithms are part of Java's set of standard methods.

So…

What can we say about Arrays of Objects?

- Arrays of objects are good for some operations
- But for other operations not so much

What can we say about Arrays of Objects?

- They are not very good when data is dynamic
- Searching is efficient only when sorted (see previous presentation)
- Keeping order is hard if new objects are inserted randomly
  - When you keep adding and removing items in the array they are hard to manage (what to do with ”holes” in the array?). Efficient sorting only works when they are sorted – but if you want to insert values randomly and keep the order you have to move a lot of bytes around.

What can we say about Arrays of Objects?

The index isn’t always a natural way to access data

Additionally, the way you refer to an element in an array is its index (its position in the array). When you refer to a city, it's more natural to give the name of the city than to say "city at position n". In a program, that means that you would probably have the program user enter a name, search the array to find the position, then use the position afterwards.

List Data Type

- Using the methods we've just seen to search, access indexes and so on we have defined the basic operations we need for a list data type
- We could put them together in a class to specify a list data type
- Java has several data types for holding groups of objects including lists
- They are in the collections interface
- **ArrayList for example implements a list basically in the way I just described**

Java Collections Framework

- Classes and interfaces in java.util
- No predefined size
- A variety of container types that are suitable for different special grouping uses

You need to know what is important for your application

Java Collections defines interfaces and classes that allow you to group objects using different internal implementations. From simple arrays to more complex structures. There are different ways to store and organize objects, all with different features and levels of efficiency. Its good to understand this and choose efficient ones for particular problems. The relationship between structures for storing data and algorithms is also a big area of study.

An Overview of the Collections Framework

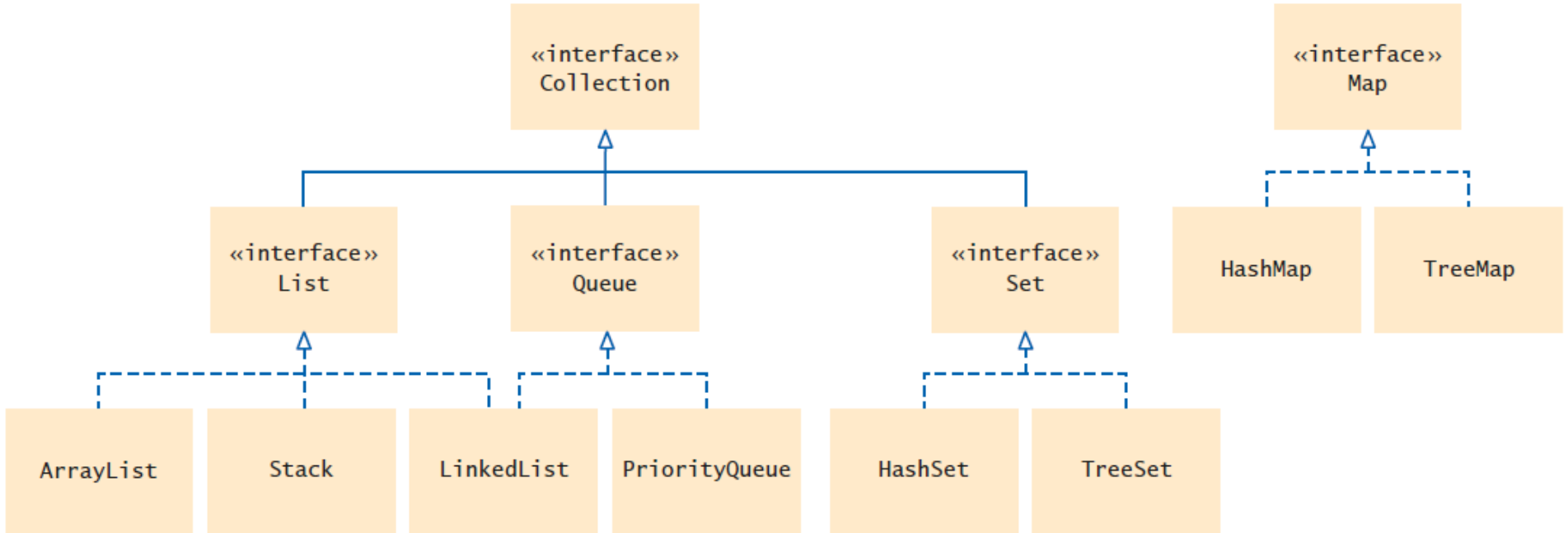
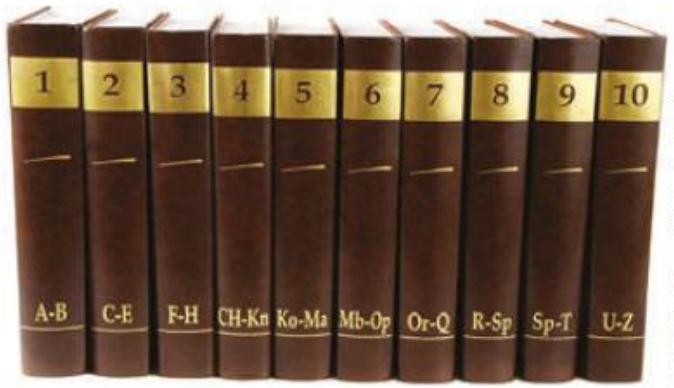


Figure 1 Interfaces and Classes in the Java Collections Framework



© Filip Fuxa/iStockphoto.

Figure 2 A List of Books


© parema/iStockphoto.

Figure 3 A Set of Books

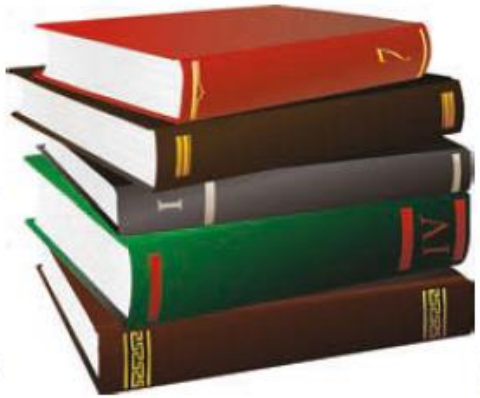

© Vladimir Trenin/iStockphoto.

Figure 4 A Stack of Books

A list is a collection that remembers the order of its elements.

A set is an unordered collection of unique elements.

A stack is a collection of elements with "last-in, first-out" retrieval.

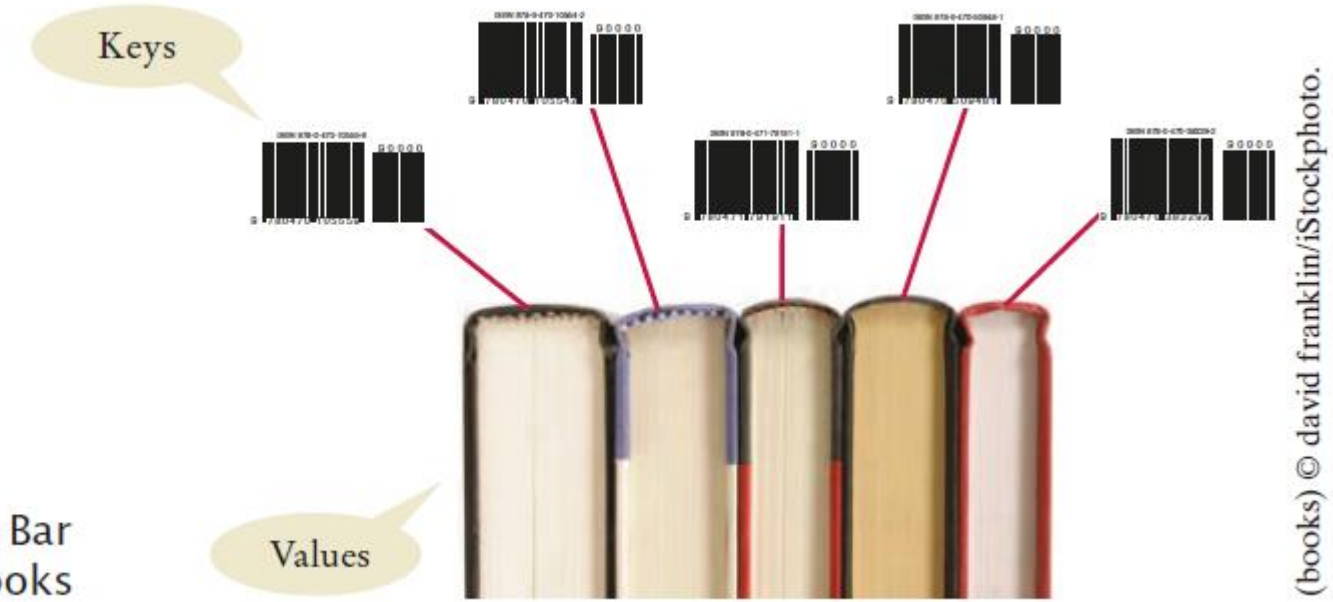


Figure 5
A Map from Bar Codes to Books

A map keeps associations between key and value objects.

Test

- A gradebook application stores a collection of quizzes. Should it use a list or a set?

Answer: A list is a better choice because the application will want to retain the order in which the quizzes were given.

- A student information system stores a collection of student records for a university. Should it use a list or a set?

Answer: A set is a better choice. There is no intrinsically useful ordering for the students. For example, the registrar's office has little use for a list of all students by their GPA. By storing them in a set, adding, removing, and finding students can be efficient.

- Why is a queue of books a better choice than a stack for organizing your required reading?

Answer: With a stack, you would always read the latest required reading, and you might never get to the oldest readings.

Table 1 The Methods of the Collection Interface

`Collection<String> coll = new ArrayList<>();`	The `ArrayList` class implements the `Collection` interface.
`coll = new TreeSet<>();`	The `TreeSet` class (Section 15.3) also implements the `Collection` interface.
`int n = coll.size();`	Gets the size of the collection. `n` is now `0`.
`coll.add("Harry");` `coll.add("Sally");`	Adds elements to the collection.
`String s = coll.toString();`	Returns a string with all elements in the collection. `s` is now `[Harry, Sally]`.
`System.out.println(coll);`	Invokes the `toString` method and prints `[Harry, Sally]`.
`coll.remove("Harry");` `boolean b = coll.remove("Tom");`	Removes an element from the collection, returning `false` if the element is not present. `b` is false.
`b = coll.contains("Sally");`	Checks whether this collection contains a given element. `b` is now `true`.

`for (String s : coll)`
`{`
`  System.out.println(s);`
`}`
	You can use the "for each" loop with any collection. This loop prints the elements on separate lines.

`Iterator<String> iter = coll.iterator();`	You use an iterator for visiting the elements in the collection (see Section 15.2.3).

An Overview of the Collections Framework

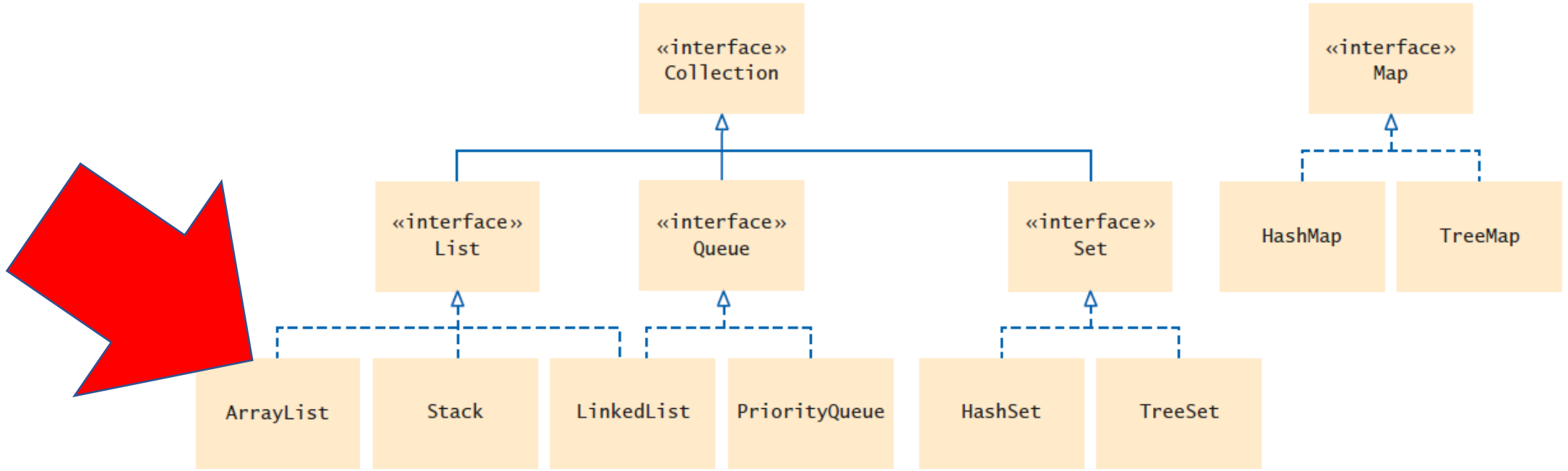


Figure 1 Interfaces and Classes in the Java Collections Framework

ArrayList is a Resizable Array

- There are different kinds of list and java.util.ArrayList is a dynamically resizable array

- `It grows automatically …`
- `… But it remains costly to resize`

  (Byte shifting still occurs)

The ArrayList that you have probably already used (if not, you may have used a Vector that is very much like an ArrayList) is such a collection. It grows automatically (which means that it automatically creates a bigger array when needed and copy the elements there – you need not do it yourself but it still happens)

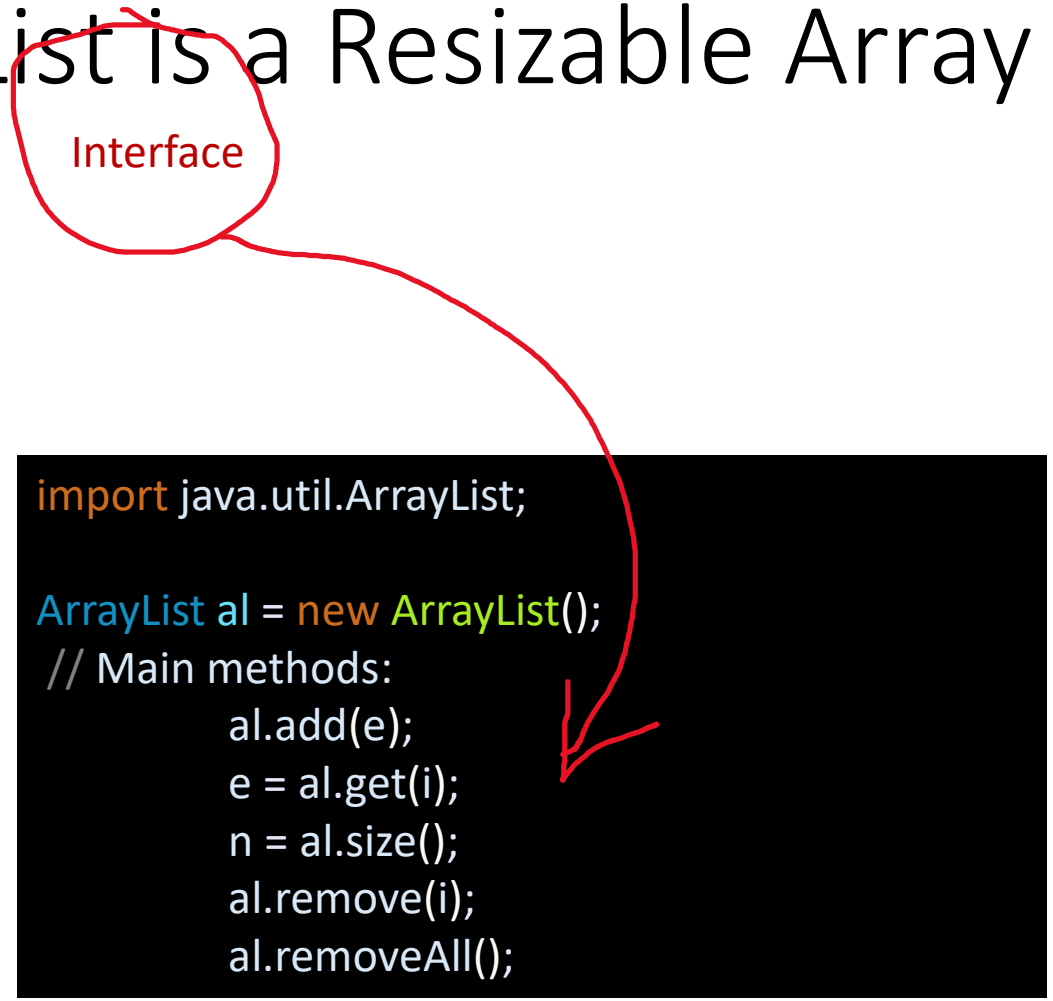
ArrayList is a Resizable Array

- You don’t use exactly the same syntax with an ArrayList as with an array, but you can basically do the same operations. Some methods apply to one element, others to all of them.

```
import java.util.ArrayList;

ArrayList al = new ArrayList();
 // Main methods:
                al.add(e);
                e = al.get(i);
                n = al.size();
                al.remove(i);
                al.removeAll();
```

careful as al[i] doesn't work!

ArrayList is a Resizable Array



"List" refers to the methods or operations that are available in the List ADT.

The array in the name refers to the implementation of the list using an array.

ArrayList is a Resizable Array

"Array" refers to how the data is actually physically stored.
You can offer the same operations but store data in different ways

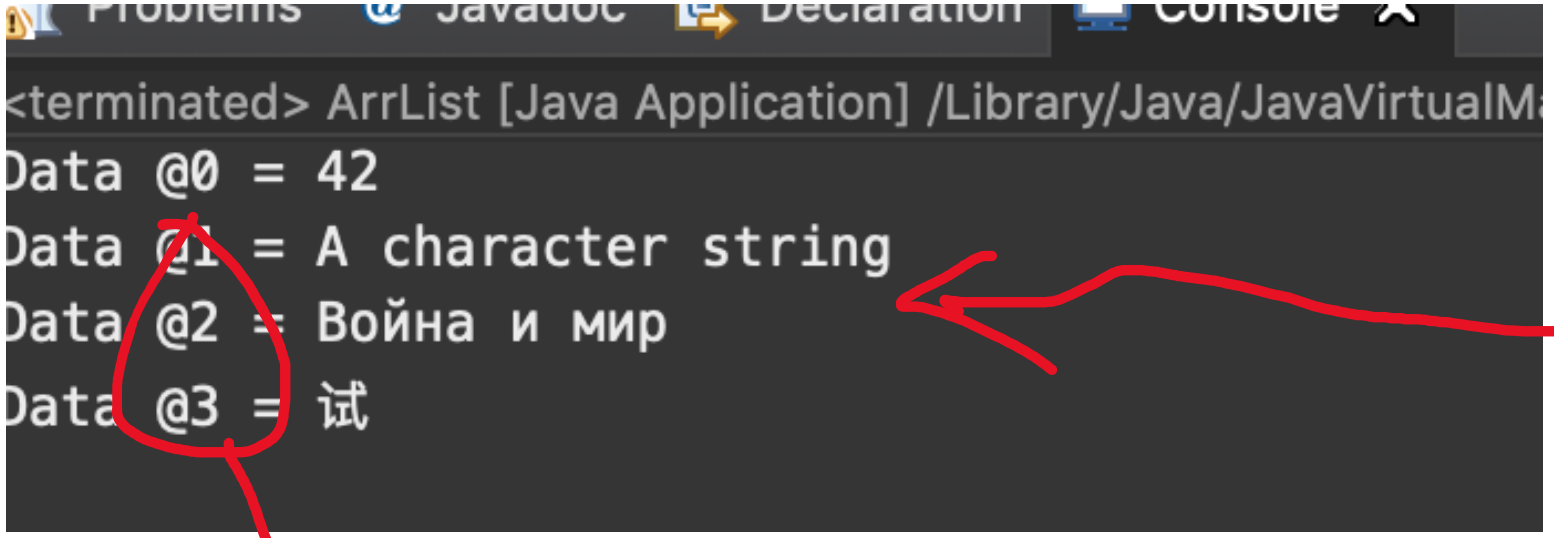
```
import java.util.ArrayList;

ArrayList al = new ArrayList();
 // Main methods:
                al.add(e);
                e = al.get(i);
                n = al.size();
                al.remove(i);
                al.removeAll();
```

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

    ArrayList al = new ArrayList();
    Random r = new Random();

    al.add(42);
    al.add("A character string");
    al.add("Война и мир");
    al.add(3.141592);
    al.add('试');

    int target = r.nextInt(al.size());
    al.remove(target);
    for (int i = 0; i < al.size(); i++) {
        System.out.println("Data @" + i + " = " + al.get(i));
    }
}
```

An ArrayList, like any collection, can store objects of any type, even of different types as here.

I'm removing a random element.

My random deletion removed the number pi. You see here that the array was rearranged and that everything is nicely displayed. But once again, a collection of different types of objects (you may have noticed autoboxing in action) is a bad idea. In a collection, the type of all items should be the same.



pi is gone

Indices are renumbered

Note that mixing very different objects is a poor programming practice, and the javac compiler isn't too happy with it. You can still run the program though.

ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized

ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized

Collections have been typed...

```
ArrayList<Object> al = new ArrayList<Object>();
```

And as you have seen in the previous example that may be typed with a generic type.

```
MyGenericList<String> nameList;
MyGenericList<Float> distanceList;
```

Collections are definitely where you want to use generics.

GENERICS

Defining templates for javac

```java
public class ArrList {

    public static void main(String [] args) {

        ArrayList<Object> al = new ArrayList<Object>();

        Random r = new Random();

        al.add(42);
        al.add("A character string");
        al.add("Война и мир");
        al.add(3.141592);
        al.add('试');

        int target = r.nextInt(al.size());
        al.remove(target);
        for (int i = 0; i < al.size(); i++) {
            System.out.println("Data @" + i + " = " + al.get(i));
        }
    }
```

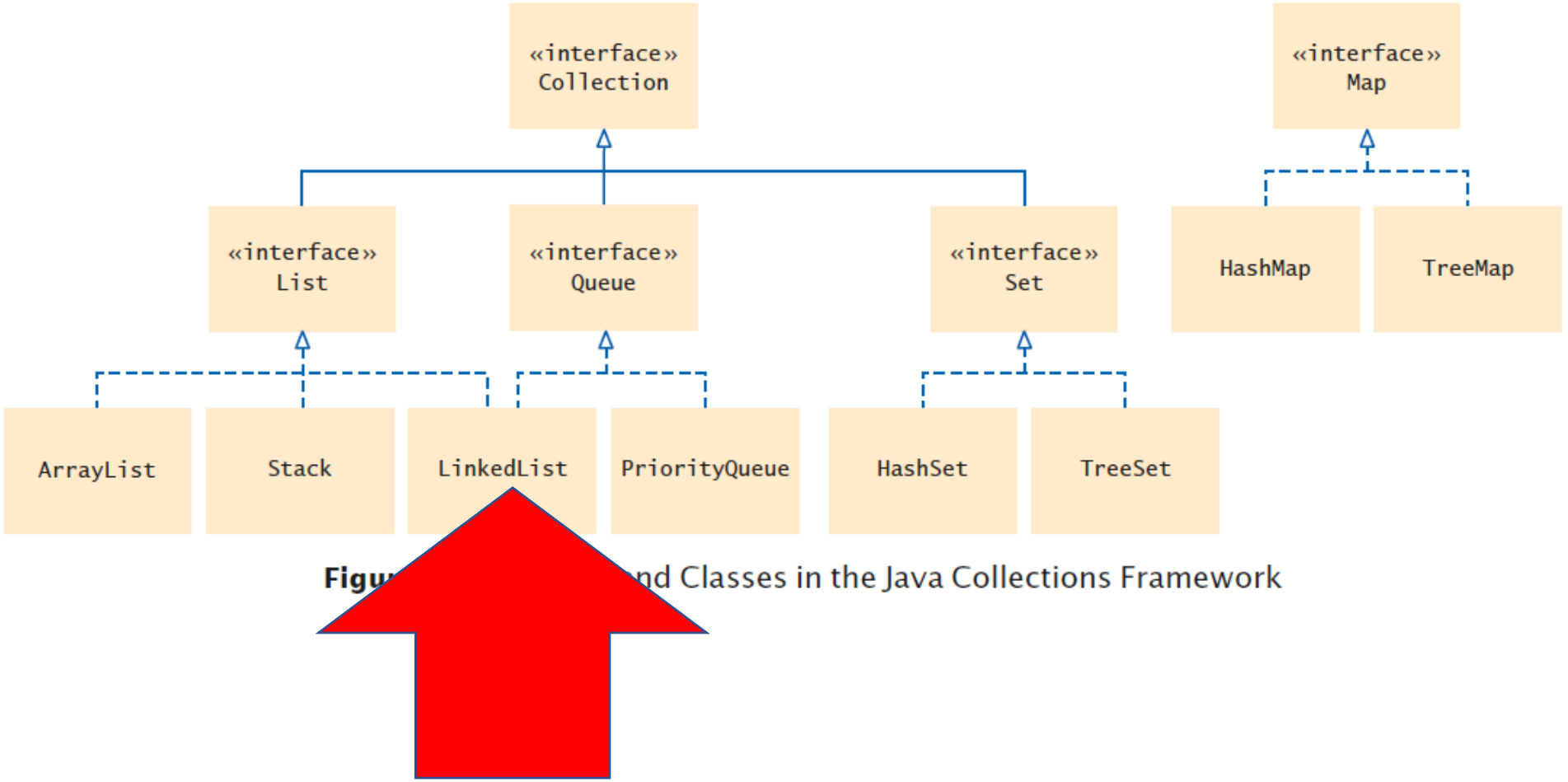
Can use the fact all items are objects to hold different object types in a string (works here as all the types have a meaningful “toString()” method implemented.

This time I deleted the integer 42

```
<terminated> ArrList [Java Application] /Library/Java
Data @0 = A character string
Data @1 = Война и мир
Data @2 = 3.141592
Data @3 = 试
```

```java
import java.util.ArrayList;

public class MyGenericList <T>{

    private ArrayList<T> _list;

    public void setElement(T o) {
        _list.add(o);
    }

    public T getElement(int i) {
        return _list.get(i);
    }
}
```

If you need to have your special wrapper around a collection, you use a generic type (here T) that you pass to the collection.

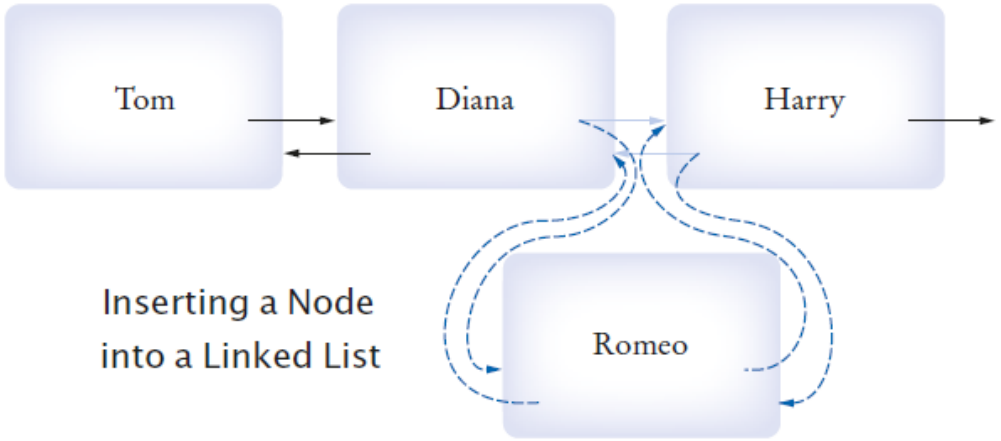


Figu nd Classes in the Java Collections Framework

Linked Lists

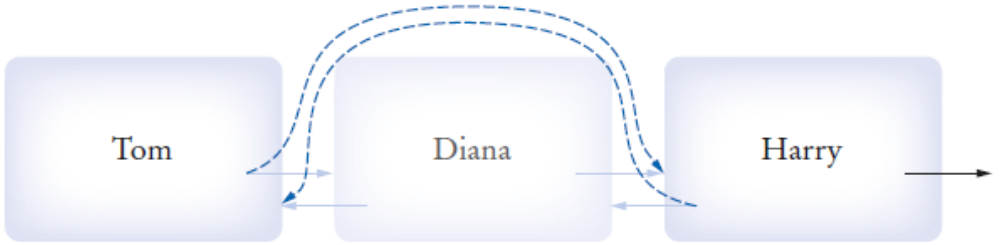
A linked list consists of a number of nodes, each of which has a reference to the next node.

Adding and removing elements at a given location in a linked list is efficient.



Inserting a Node into a Linked List

Visiting the elements of a linked list in sequential order is efficient, but random access is not.



Removing a Node from a Linked List

Linked Lists

Use linked lists when the efficiency of inserting or removing elements is important (avoid leaving “holes”) and you rarely need element access in random order.

As we saw use Generic class Specify type of elements in angle brackets: eg
LinkedList<Product>Package

Table 2 Working with Linked Lists

`LinkedList<String> list = new LinkedList<>();`	An empty list.
`list.addLast("Harry");`	Adds an element to the end of the list. Same as `add`.
`list.addFirst("Sally");`	Adds an element to the beginning of the list. `list` is now `[Sally, Harry]`.
`list.getFirst();`	Gets the element stored at the beginning of the list; here `"Sally"`.
`list.getLast();`	Gets the element stored at the end of the list; here `"Harry"`.
`String removed = list.removeFirst();`	Removes the first element of the list and returns it. `removed` is `"Sally"` and `list` is `[Harry]`. Use `removeLast` to remove the last element.
`ListIterator<String> iter = list.listIterator()`	Provides an iterator for visiting all list elements (see Table 3 on page 698).

List Iterator

Initially points before the first element.
Move the position with next method:

```
if (iterator.hasNext()) {
iterator.next();
}
```

The next method returns the element that the iterator is passing.

The return type of the next method matches the list iterator's type parameter.

List Iterator

```
while (iterator.hasNext()) {
   String name = iterator.next();
  //Do something with name
}
// To use the “for each” loop:
for (String name : employeeNames) {
  //Do something with name
}
```

List Iterator The nodes of the LinkedList class store two links: One to the next element One to the previous one Called a doubly-linked list
To move the list position backwards,

```
while (iterator.hasPrevious()) {
   String name = iterator.previous();
  //Do something with name
}
```

The iterator can also be used to add and remove elements eg iterator.add(element)

```
LinkedList<String> employeeNames = . . .;
ListIterator<String> iterator = employeeNames.listIterator();
```

Table 3 Methods of the Iterator and ListIterator Interfaces

`String s = iter.next();`	Assume that `iter` points to the beginning of the list `[Sally]` before calling `next`. After the call, `s` is `"Sally"` and the iterator points to the end.
`iter.previous();` `iter.set("Juliet");`	The `set` method updates the last element returned by `next` or `previous`. The list is now `[Juliet]`.
`iter.hasNext()`	Returns `false` because the iterator is at the end of the collection.

`if (iter.hasPrevious())`
`{`
`  s = iter.previous();`
`}`
	`hasPrevious` returns `true` because the iterator is not at the beginning of the list. `previous` and `hasPrevious` are `ListIterator` methods.

`iter.add("Diana");`	Adds an element before the iterator position (`ListIterator` only). The list is now `[Diana, Juliet]`.
`iter.next();` `iter.remove();`	`remove` removes the last element returned by `next` or `previous`. The list is now `[Diana]`.

Use a list iterator to access elements inside a linked list.
Encapsulates a position anywhere inside the linked list.
Think of an iterator as pointing between two elements:
Analogy: like the cursor in a word processor points between two characters

An Overview of the Collections Framework

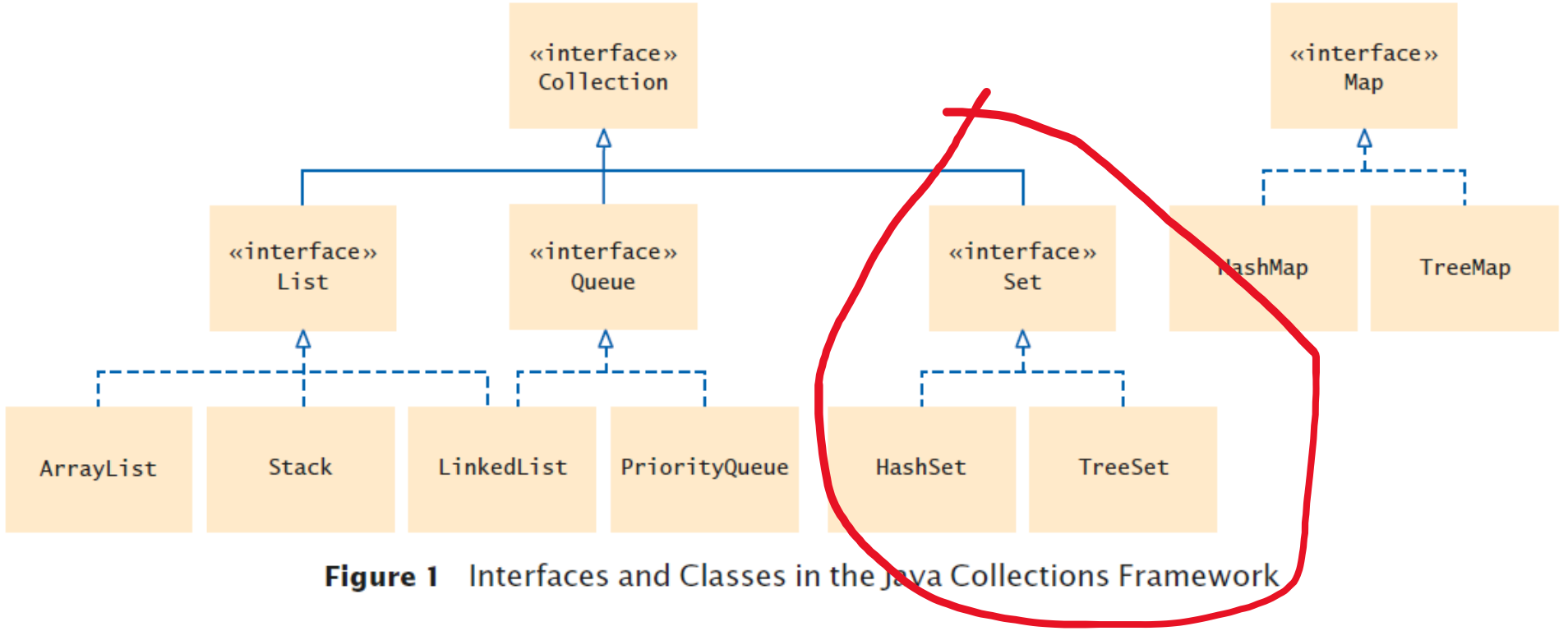


Figure 1 Interfaces and Classes in the Java Collections Framework

The `HashSet` and `TreeSet` classes both implement the `Set` interface.

Set implementations arrange the elements so that they can locate them quickly.

You can form hash sets holding objects of type `String`, `Integer`, `Double`, `Point`, `Rectangle`, or `Color`.

You can form tree sets for any class that implements the `Comparable` interface, such as `String` or `Integer`.

When you construct a `HashSet` or `TreeSet`, store the reference in a `Set` variable.

```
Set<String> names = new HashSet<>();
```

or

```
Set<String> names = new TreeSet<>();
```

Tree set or Hashset?

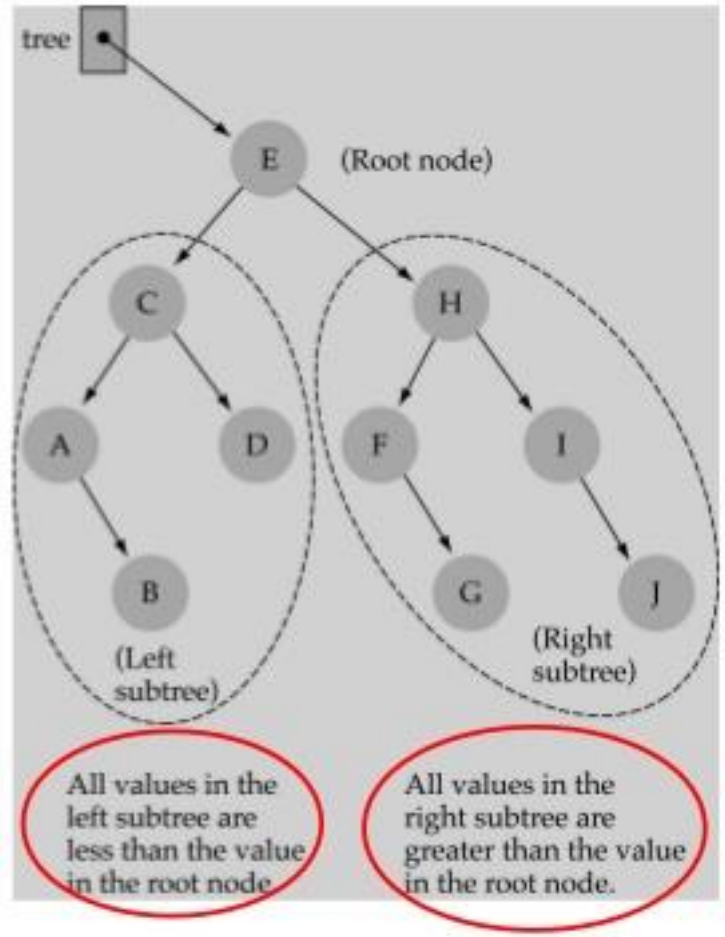
Use a TreeSet if you want to visit the set's elements in sorted order. Otherwise choose a HashSet.It is a bit more efficient.

A good hash function (certainly different) hash values for each object so that they are scattered about in a hash table.

Binary Search Trees

- **Binary Search Tree Property**:

  The value stored at a node is *greater* than the value stored at its left child and *less* than the value stored at its right child



Set<String> names
= new HashSet<>();

Table 4 Working with Sets

`Set<String> names;`	Use the interface type for variable declarations.
`names = new HashSet<>();`	Use a `TreeSet` if you need to visit the elements in sorted order.
`names.add("Romeo");`	Now `names.size()` is 1.
`names.add("Fred");`	Now `names.size()` is 2.
`names.add("Romeo");`	`names.size()` is still 2. You can't add duplicates.
`if (names.contains("Fred"))`	The `contains` method checks whether a value is contained in the set. In this case, the method returns `true`.
`System.out.println(names);`	Prints the set in the format `[Fred, Romeo]`. The elements need not be shown in the order in which they were inserted.
`for (String name : names) { . . . }`	Use this loop to visit all elements of a set.
`names.remove("Romeo");`	Now `names.size()` is 1.
`names.remove("Juliet");`	It is not an error to remove an element that is not present. The method call has no effect.

Iterator<String> iter = names.iterator(); while (iter.hasNext())

An Overview of the Collections Framework

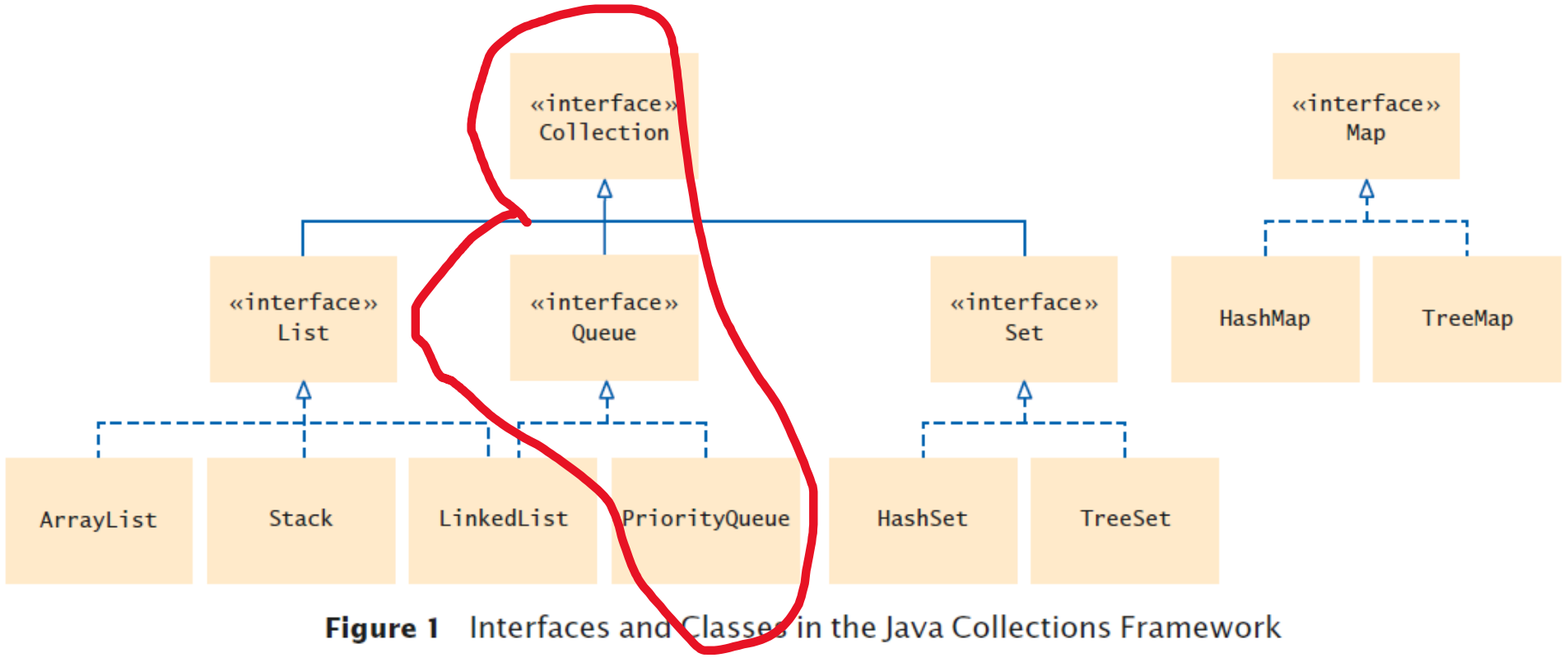


Figure 1 Interfaces and Classes in the Java Collections Framework

Table 8 Working with Queues

`Queue<Integer> q = new LinkedList<>();`	The `LinkedList` class implements the `Queue` interface.

`q.add(1);`
`q.add(2);`
`q.add(3);`
	Adds to the tail of the queue; `q` is now `[1, 2, 3]`.

`int head = q.remove();`	Removes the head of the queue; `head` is set to 1 and `q` is `[2, 3]`.
`head = q.peek();`	Gets the head of the queue without removing it; `head` is set to 2.

The LinkedList class implements the Queue interface. When you need a queue, initialize a Queue variable with a LinkedList object:

```
Queue<String> q = new LinkedList<>();
q.add("A");
q.add("B");
q.add("C");
while (q.size() > 0) {
System.out.print(q.remove() + " ");
}
```

Table 9 Working with Priority Queues

`PriorityQueue<Integer> q = new PriorityQueue<>();`	This priority queue holds `Integer` objects. In practice, you would use objects that describe tasks.
`q.add(3); q.add(1); q.add(2);`	Adds values to the priority queue.
`int first = q.remove();` `int second = q.remove();`	Each call to `remove` removes the most urgent item: `first` is set to 1, `second` to 2.
`int next = q.peek();`	Gets the smallest value in the priority queue without removing it.

Because the priority queue needs to be able to tell which element is the smallest, the added elements should belong to a class that implements the `Comparable` interface. Thus, each removal operation extracts the *minimum* element from the queue.

An Overview of the Collections Framework

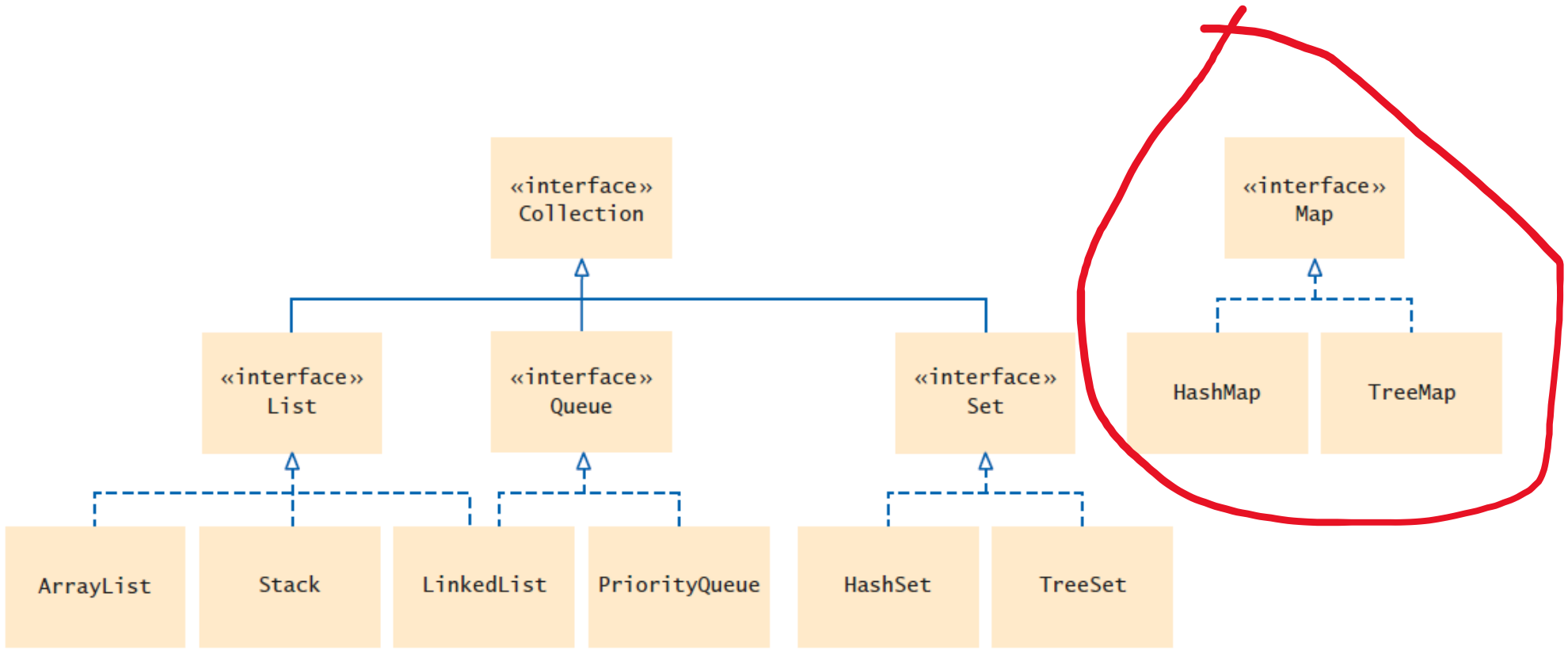
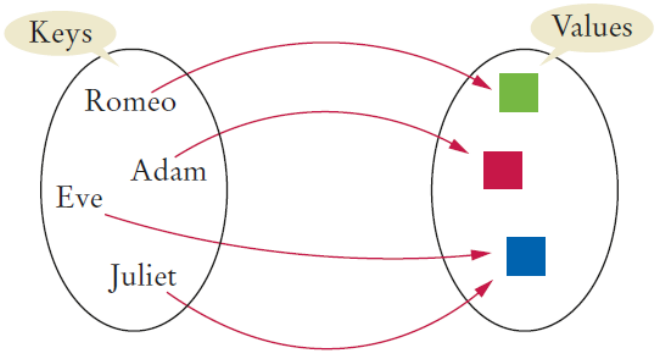


Figure 1 Interfaces and Classes in the Java Collections Framework

Maps

The HashMap and TreeMap classes both implement the Map interface.



A Map

In a *TreeMap*, the key/value associations are stored in a sorted tree, in which they are sorted according to their `keys`. For this to work, it must be possible to compare the keys to one another.

This means either that the keys must implement the interface *Comparable<K>*, or that a *Comparator* must be provided for comparing keys.
(The *Comparator* can be provided as a parameter to the *TreeMap* constructor.)

Note that in a *TreeMap*, as in a *TreeSet*, the `compareTo()` (or `compare()`) method is used to decide whether two keys are to be considered the same.

After constructing a HashMap or TreeMap, you can store the reference to the map object in a Map reference:

```
Map<String, Color> favoriteColors = new HashMap<>();
```

Map<String, Color> favoriteColors = new HashMap<>();

Suppose that `map` is a variable of type *Map<K,V>* for some specific types *K* and *V*.
Then the following are some of the methods that are defined for `map`:

- `map.get(key)` — returns the object of type *V* that is associated by the map to the `key`. If the map does not associate any value with `key`, then the return value is `null`. Note that it's also possible for the return value to be `null` when the map explicitly associates the value `null` with the key. Referring to "`map.get(key)`" is similar to referring to "`A[key]`" for an array `A`. (But note that there is nothing like an *IndexOutOfBoundsException* for maps.)

Color julietsFavoriteColor =
favoriteColors.get("Juliet");

- `map.put(key,value)` — Associates the specified `value` with the specified `key`, where `key` must be of type *K* and `value` must be of type *V*. If the map already associated some other value with the key, then the new value replaces the old one. This is similar to the command "`A[key] = value`" for an array.

favoriteColors.put("Juliet", Color.BLUE);

- `map.putAll(map2)` — if `map2` is another map of type *Map<K,V>*, this copies all the associations from `map2` into map.
- `map.remove(key)` — if `map` associates a value to the specified `key`, that association is removed from the map.

favoriteColors.remove("Juliet");

- `map.containsKey(key)` — returns a boolean value that is `true` if the map associates some value to the specified `key`.
- `map.containsValue(value)` — returns a boolean value that is `true` if the map associates the specified `value` to some key.

Maps

Table 5 Working with Maps	
`Map<String, Integer> scores;`	Keys are strings, values are `Integer` wrappers. Use the interface type for variable declarations.
`scores = new TreeMap<>();`	Use a `HashMap` if you don't need to visit the keys in sorted order.
`scores.put("Harry", 90);` `scores.put("Sally", 95);`	Adds keys and values to the map.
`scores.put("Sally", 100);`	Modifies the value of an existing key.
`int n = scores.get("Sally");` `Integer n2 = scores.get("Diana");`	Gets the value associated with a key, or `null` if the key is not present. `n` is 100, `n2` is `null`.
`System.out.println(scores);`	Prints `scores.toString()`, a string of the form `{Harry=90, Sally=100}`

`for (String key : scores.keySet()) {`
`  Integer value = scores.get(key);`
`  . . .`
`}`
	Iterates through all map keys and values.

`scores.remove("Sally");`	Removes the key and value.

If `map` is a variable of type *Map<K,V>*, then

The value returned by `map.keySet()` is a "view" of keys in the map implements the *Set<K>* interface

```
Set<String> keySet = m.keySet();
for (String key : keySet){
Color value = m.get(key);
System.out.println(key + "->" + value);
}
```

`map.values()` returns an object of type *Collection<V>* that contains all the values from the associations that are stored in the map. The return value is a *Collection* rather than a *Set* because it can contain duplicate elements.

One of the things that you can do with a *Set* is get an *Iterator* for it and use the iterator to visit each of the elements of the set in turn.

`map.entrySet()` returns a set that contains all the associations from the map.

The elements in the set are objects of type *Map.Entry<K,V>*.

The return type is written as *Set<Map.Entry<K,V>>*.

Each `Map.Entry` object contains one key/value pair, and defines methods `getKey()` and `getValue()` for retrieving the key and the value.

Pop quiz

- What is the difference between a set and a map?

Answer: A set stores elements. A map stores associations between keys and values.

- Why is the collection of the keys of a map a set and not a list?

Answer: The ordering does not matter, and you cannot have duplicates

- Why is the collection of the values of a map not a set?

Answer: Because it might have duplicates.

Table 7 Working with Stacks

`Stack<Integer> s = new Stack<>();`	Constructs an empty stack.

`s.push(1);`
`s.push(2);`
`s.push(3);`
	Adds to the top of the stack; `s` is now `[1, 2, 3]`. (Following the `toString` method of the `Stack` class, we show the top of the stack at the end.)

`int top = s.pop();`	Removes the top of the stack; `top` is set to 3 and `s` is now `[1, 2]`.
`head = s.peek();`	Gets the top of the stack without removing it; `head` is set to 2.

```
if (!s.empty()) …
if (s.size() > 0) …
```