

Introduction to computer science II

CMPE 212

Who Am I?

- ▶ Dr. Burton Ma
- ▶ office
 - ▶ Goodwin 754
- ▶ office hours are virtual on Teams
 - ▶ TBA
- ▶ email
 - ▶ include CMPE212 in subject line so I know what course you are in
 - ▶ burton.ma@queensu.ca

Teaching assistants

- ▶ TBA
- ▶ contact information and office hours will be posted on onq

Course information

- ▶ everything will be on onq
- ▶ lectures are live streamed and recorded

Textbooks

- ▶ course notes provided as Jupyter notebooks
 - ▶ see onq for instructions, or online at:
 - ▶ <https://mybinder.org/v2/gh/burtonma/CISC124-Fall-2021/main?filepath=toc.ipynb>
- ▶ library has excellent online books:
 - ▶ Introduction to Programming in Java: An Interdisciplinary Approach, 2nd ed (excellent for its breadth of exercises, but weak or lacking in some aspects of object-oriented programming)
 - ▶ https://ocul-qu.primo.exlibrisgroup.com/permalink/o1OCUL_QU/1mijubc/alma9952532745005158
 - ▶ Effective Java, 2nd ed (3rd edition is actually available if you look for it after following the link)
 - ▶ https://ocul-qu.primo.exlibrisgroup.com/permalink/o1OCUL_QU/sk7he5/cdi_askewsholts_vlebooks_9780134686073

Labs

- ▶ virtual (even after Reading Week)
 - ▶ will be posted on onq as they become available
- ▶ can be completed any time before due date
 - ▶ i.e., your scheduled lab times are unused

Grading

- ▶ TBA
 - ▶ will be posted on onq later this week
 - ▶ assignments approximately every 2 weeks
 - ▶ weekly labs
 - ▶ quiz or quizzes
 - ▶ exam

Assignments

- ▶ to be done individually, submitted via onq
- ▶ solution posted 72 hours after assignment due date
- ▶ late submission policy
 - ▶ not accepted after 72 hours after the posted due date
 - ▶ this includes students with accommodations (contact me for alternate arrangements)
 - ▶ lose 10% of the total marks for the assignment for each 24 hour period late

Quizzes

- ▶ format is not yet fixed, but likely to be online on onq during your regularly scheduled lecture

Exam

- ▶ to be determined

What is this course about?

CMPE 212: Introduction to computing science II

Introduction to object-oriented design, architecture, and programming. Use of packages, class libraries, and interfaces. Encapsulation and representational abstraction. Inheritance. Polymorphic programming. Exception handling. Iterators. Introduction to a class design notation. Applications in various areas.

- ▶ learn about object-oriented programming using Java

What is object-oriented programming?

- ▶ one of the most widely used programming paradigms
- ▶ programming paradigms
 - ▶ structured programming
 - ▶ like what you learned in APSC 142, ELEC 278
 - ▶ control flow defined by loops, conditionals, function calls
 - ▶ object-oriented programming (OOP)
 - ▶ running program made up of objects that have state (information) and behavior (methods)
 - ▶ objects interact with one another by passing messages (calling methods)
 - ▶ many others

What is Java?

- ▶ a general purpose, cross platform, OOP language
- ▶ created during early 1990s to address issues with the dominant languages at the time (C/C++)
- ▶ primary goals
 1. simple, object-oriented, and familiar.
 2. robust and secure.
 3. architecture-neutral and portable
 4. execute with high performance
 5. interpreted, threaded, and dynamic

Where is Java used?

- ▶ enterprise software
- ▶ server-side
- ▶ big data
- ▶ embedded devices
- ▶ some scientific applications
- ▶ some video games

- ▶ see <https://blogs.oracle.com/javamagazine/the-top-25-greatest-java-apps-ever-written>

Using Java this course

- ▶ instructions on onq
 - ▶ install a Java Development Kit (JDK)
 - ▶ Oracle JDK 17 recommended
 - ▶ install eclipse for Java programmers integrated development environment (IDE)
 - ▶ other IDEs are fine but I cannot offer support for them

Jupyter notebooks

- ▶ installation instructions are on onq
 - ▶ but installation is not always easy
 - ▶ Python code cells not working
 - ▶ students using non-English character set on their computers will probably not be able to run the code cells in the notebook
- ▶ demo here

From C to Java

CMPE 212

A C program starts running from the function **main**

- ▶ there are two legal forms of the main function in modern C
- ▶ plus compiler writers may allow their own versions of **main**

```
#include <stdio.h>
int main(void) {
    puts("Hello, world!");
}
```

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    puts("Hello, world!");
}
```

A Java program starts running from a **main** method

- ▶ a **main** method must belong to a class (or interface) and it must have a specific form
- ▶ exactly two allowed forms of the **main** method

```
public class HelloWorld {  
    public static void main(String... args) {  
        System.out.println("Hello, world!");  
    }  
}
```

A Java program starts running from a **main** method

- ▶ there are two legal forms of **main**
 - ▶ see previous slide for one of the two forms
 - ▶ the second form is shown below

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

A Java class name starts with an uppercase letter and contains no spaces

- ▶ use upper **CamelCase** for multiword class names (first letter of every word of the name is capitalized)
- ▶ avoid underscores except in unusual cases

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

Java statements end with a semi-colon

- ▶ not precisely true, but the easiest way to convey the idea at this point

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

C has no special type for strings (sequences of characters).

- ▶ uses a null-terminated array of **char** for strings

```
char s[] = "Hello, world!";
```

Java has a separate type that represents strings.

- ▶ each unique string is represented by a different **String** object

```
String s = "Hello, world!";  
s = "Hej verden!";  
s = "Molo lizwe!";
```

Performing simple tasks with strings in C is tedious and error prone.

- ▶ form example, when joining, or *concatenating*, strings, need to make sure that the array has sufficient capacity

```
char s[14] = "Hello";
char t[] = ", world!";
// join s and t storing result in s
strcat(s, t);
```

Java hides memory allocation from the programmer.
Concatenating strings can be done using the + operator.

```
String s = "Hello";
String t = ", world!";
String u = s + t;
```

Printing text in C is usually performed using **printf**

- ▶ printing different types requires a different conversion flag

```
int num = 212;  
char subj[] = "CMPE";  
printf("%s %d\n", subj, num);
```

Java's **print** and **println** methods can handle any Java type.

- ▶ **print** does not go to the next line after printing

```
int num = 212;  
String subj = "CMPE";  
System.out.print(subj);  
System.out.print(" ");  
System.out.println(num);  
  
// or use String concatenation  
System.out.println(subj + " " + num);
```

C variables have a type

- ▶ the type is declared along with the variable name
- ▶ the type cannot be changed
- ▶ variable can hold values only of the declared type

```
char s[] = "Hello, world!";
int t = 123;
int arr[3];
arr[0] = 0;
arr[1] = 1;
arr[2] = 2;
```

Java variables have a type

- ▶ the type is declared along with the variable name
- ▶ the type cannot be changed
- ▶ variable can hold values only of the declared type

```
String s = "Hello, world!";
```

```
int t = 123;
```

```
int[] arr = new int[3];
```

```
arr[0] = 0;
```

```
arr[1] = 1;
```

```
arr[2] = 2;
```

Java variable names begin with a lowercase letter and contain no spaces

- ▶ use lower **camelCase** for multiword variable names
 - ▶ first letter of every word after the first is capitalized

```
String s = "Hello, world!";  
int t = 123;  
int[] anArray = new int[3];  
anArray[0] = 0;  
anArray[1] = 1;  
anArray[2] = 2;
```

A line ending comment starts with `//` in Java.

- ▶ identical to C

```
// initial investment
double p = 100;

// annual interest rate in percent
double r = 1.25;

double final_value =
    p * Math.pow((1 + r / 100.0), 10); // after 10 years
```

A multiline Java comment starts with `/*` and ends with `*/`

- ▶ identical to C

```
/* initial investment and  
   annual interest rate in percent */  
double p = 100;  
double r = 1.25;  
  
double finalValue = /* hey, I can put a comment here */  
    p * Math.pow(1 + r, 10);           // after 10 years
```

Prior to C99, there was no standard type for true/false values.

- ▶ traditionally, integer **0** represents false and any non-zero value is converted to true

```
int tf = 0;  
if (tf) {  
    puts("true");  
}  
else {  
    puts("false");  
}
```

The true/false type in Java is called **boolean**

- ▶ only possible values are **true** and **false**
- ▶ there is no built-in way to convert a non-**boolean** value to a **boolean** value

```
boolean tf = false;  
if (tf) {  
    System.out.println("true");  
}  
else {  
    System.out.println("false");  
}  
  
// or just  
// System.out.println(tf);
```

Built-in C integer types and literals

```
char                  c  = 'a';
signed char          h  = -3;
unsigned char        uh = 2U;
short int            s  = -1;
unsigned short int   us = 2U;
int                  i  = -100;
unsigned int          ui = 200U;
long int              n  = -5L;
unsigned long int    un = 5UL;
long long int        g  = -3LL;
unsigned long long int ug = 7ULL;
```

Built-in Java integer types and literals

- ▶ no unsigned types in Java
- ▶ some type names changed compared to C

byte	<code>y = 1;</code>
char	<code>c = 'a';</code>
short	<code>s = -1;</code>
int	<code>i = -100;</code>
long	<code>n = -5L;</code>

Built-in C floating-point types and literals

```
float          f = 1.5F;  
double         d = 3.1415;  
long double dd = 2.71828L;  
  
// scientific notation  
f = 1.5e1F;    // 1.5 times 10 to the power 1  
d = 1e100;      // 1   times 10 to the power 100  
dd = 1e100L;    // 1   times 10 to the power 100
```

Built-in Java floating-point types and literals

- ▶ no **long double** in Java

```
float          f = 1.5F;  
double        d = 3.1415;
```

```
// scientific notation  
f = 1.5e1F;    // 1.5 times 10 to the power 1  
d = 1e100;      // 1   times 10 to the power 100
```

The Java Standard mandates the size of each type.

- ▶ each type occupies a specified amount of memory
- ▶ each type has a specified minimum and maximum value

Type	Memory	Minimum	Maximum
byte	8-bit signed	-128	127
char	8-bit unsigned	0	65,535
short	16-bit signed	-32,768	32,767
int	32-bit signed	-2^{31}	$2^{31} - 1$
long	64-bit signed	-2^{63}	$2^{63} - 1$
float	32-bit IEEE754	$\sim -3.4 \times 10^{38}$	$\sim 3.4 \times 10^{38}$
double	64-bit IEEE754	$\sim -1.8 \times 10^{308}$	$\sim 1.8 \times 10^{308}$

No pointer types in Java

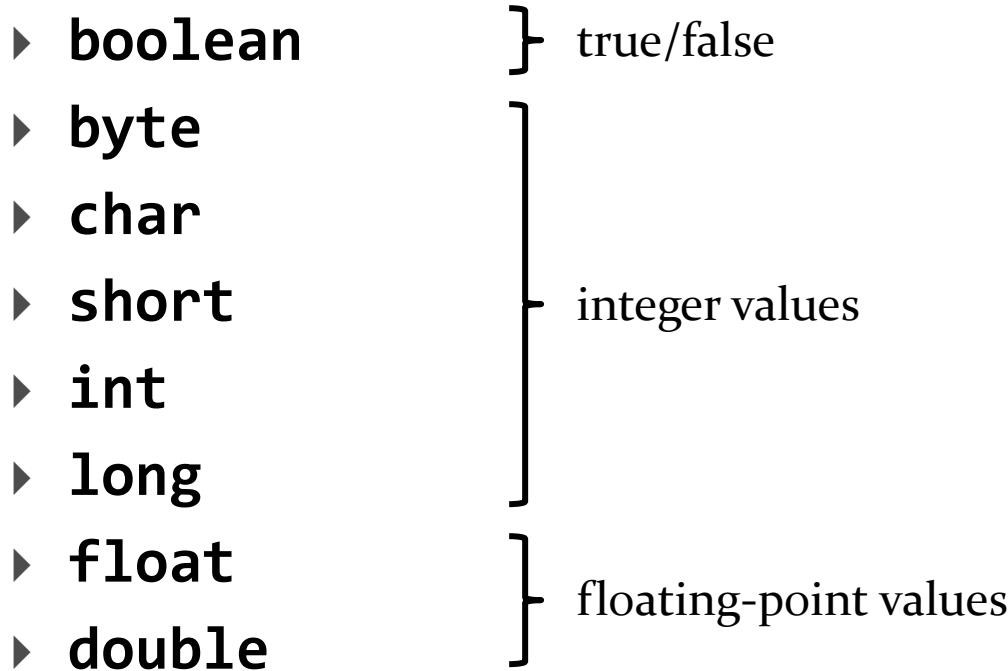
- ▶ Java has no mechanism for accessing memory directly
 - ▶ therefore, there are no pointer types
- ▶ Java programmers generally are not concerned with allocating memory for objects
 - ▶ except for arrays where the size of the array has to be specified
 - ▶ de-allocating memory used by objects that are no longer needed is done automatically by the garbage collector

Primitive versus reference types

- ▶ Java has reference types
 - ▶ any type that begins with a capital letter is a reference type
 - ▶ e.g., **String**, **List**, **ArrayList**
 - ▶ classes are user-defined reference types
 - ▶ that is why their names should always begin with a capital letter
- ▶ Java has primitive types
 - ▶ any type that begins with a lowercase letter is a primitive type
 - ▶ unless you are a jerk and name your classes starting with a lowercase letter

Primitive types

- ▶ eight primitive types in Java



Reference variables store references to objects

- ▶ a reference is some value that allows the JVM to find an object in memory
 - ▶ often implemented as a pointer, i.e., a reference is the memory address of an object

```
// s is a reference to a String object
//      i.e., s stores the memory address of the object
//      that represents the String "hello"
String s = "hello";

// s now refers to a different String object
//      the previous String object is no longer available
//      to us
s = "goodbye";
```

Primitive type variables actually store a primitive value

```
int x = 1;           // x stores the value 1
double y = 1.0;     // y stores the value 1.0
```

Primitive types

- ▶ all of the primitive types occupy a defined fixed amount of memory
- ▶ different primitive types occupy different amounts of memory
- ▶ what is the implication of this?

int is the usual choice for routine calculations involving integer values

```
int lo = Integer.MIN_VALUE;    // -2147483648  
int hi = Integer.MAX_VALUE;    // 2147483647
```

An **int** literal is any number written without using a decimal and not ending in **L** or **l** (lower case ell)

- ▶ it is a compile-time error if the numeric value is outside the range that an **int** can represent

```
int x = -1;
```

```
int y = 99 - 98 + 1 - 0;
```

```
// underscores allowed between digits
```

```
// (not allowed at beginning or end)
```

```
int z = 1_000_000;
```

```
// binary, octal, and hexadecimal allowed but not
```

```
// relevant for our purposes
```

An arithmetic operation involving two **int** values always produces an **int** value

- ▶ division is truncating division (throw away the fractional part of the quotient)

```
int x = 1;  
int y = 3;  
int z = 13;  
int result = x + y;           // 4  
result = x - y;             // -2  
result = x * y;             // 3  
result = x / y;             // 0  
result = z / y;             // 4  
result = (x + y + z) / 3;   // 5
```

An arithmetic calculation involving two `int` values that produces a value greater than `Integer.MAX_VALUE` wraps around to the other end of the range

```
int x = Integer.MAX_VALUE;  
int y = x + 1;                      // Integer.MIN_VALUE  
y = x + 2;                          // Integer.MIN_VALUE + 1  
y = x + 3;                          // Integer.MIN_VALUE + 2
```

An arithmetic calculation involving two `int` values that produces a value less than `Integer.MIN_VALUE` wraps around to the other end of the range

```
int x = Integer.MIN_VALUE;  
int y = x - 1;                      // Integer.MAX_VALUE  
y = x - 2;                          // Integer.MAX_VALUE - 1  
y = x - 3;                          // Integer.MAX_VALUE - 2
```

Integer overflow

- ▶ *integer overflow* occurs when an arithmetic calculation with an integer type results in a value that is outside the range that can be represented by the type
- ▶ in Java, wrapping occurs, but different results occur in other programming languages
- ▶ cause of many infamous events:
 - ▶ Ariane 5 [https://en.wikipedia.org/wiki/Cluster_\(spacecraft\)#Launch_failure](https://en.wikipedia.org/wiki/Cluster_(spacecraft)#Launch_failure)
 - ▶ Deep Impact [https://en.wikipedia.org/wiki/Deep_Impact_\(spacecraft\)#Contact_lost_and_end_of_mission](https://en.wikipedia.org/wiki/Deep_Impact_(spacecraft)#Contact_lost_and_end_of_mission)
 - ▶ Therac 25 <https://en.wikipedia.org/wiki/Therac-25>
 - ▶ GPS week rollover https://en.wikipedia.org/wiki/GPS_week_number_rollover
 - ▶ Year 2000 problem https://en.wikipedia.org/wiki/Year_2000_problem
 - ▶ Year 2038 problem https://en.wikipedia.org/wiki/Year_2038_problem

double is the usual choice for routine calculations involving floating-point values

```
double hi = Double.MAX_VALUE;    // a very big value
double lo = -hi;

// the smallest positive value
double tiny = Double.MIN_VALUE;
```

A double literal is any number written using a decimal and not ending in F or f

- ▶ it is a compile-time error if the numeric value is outside the range that an **double** can represent

```
double x = 1.0;
```

```
double y = 2.;
```

```
// underscores allowed between digits
```

```
// (not allowed at beginning or end)
```

```
double z = 1_000_000.99;
```

```
// scientific notation
```

```
double alsoZ = 1.00000099e6;
```

An arithmetic operation involving two **double** values
always produces a **double** value

```
double x = 1.0;  
double y = 3.0;  
double z = 13.0;  
double result = x + y;           // 4.0  
result = x - y;                 // -2.0  
result = x * y;                 // 3.0  
result = x / y;                 // 0.33...  
result = z / y;                 // 4.33...  
result = (x + y + z) / 3;       // 5.66...
```

Unlike the integer types, floating-point computations do not wrap when values exceed the range of **double**

- values saturate at **Double.POSITIVE_INFINITY** and **Double.NEGATIVE_INFINITY**

```
double x = 1e200 * 1e200;      // Double.POSITIVE_INFINITY
double y = -1e200 * 1e200;     // Double.NEGATIVE_INFINITY
```

C has user-defined functions

- ▶ procedures that can be called to perform a specific task, e.g., find the max of two integer values
- ▶ functions can be declared in a header file

```
#ifndef MAX2_H
#define MAX2_H

// max2.h

int max2(int a, int b);

#endif // MAX_2
```

C has user-defined functions

- ▶ functions can be defined in a source code file

```
#include "max2.h"

// max2.c

int max2(int a, int b) {
    int twice_max = a + b + abs(a - b);
    return twice_max / 2;
}
```

To call a function, import the appropriate header file, call the function passing the appropriate arguments, and store the return value (if any):

```
#include "max2.h"

// my_program.c

int main(void) {
    int x = max2(5, -3);
    // do something with z here...
}
```

Java has no functions

- ▶ the closest analog is a **public static** method
- ▶ a method is similar to a function but it must be defined inside of a class (or interface)

```
public class Lecture2 {  
  
    public static max2(int a, int b) {  
        int twiceMax = a + b + Math.abs(a - b);  
        return twiceMax / 2;  
    }  
}  
  
// continued on next slide
```

To call the **max2** method, a method in another class must import the **Lecture2** class and then call the **max2** method using the syntax

Lecture2.max2(arg1, arg2)

```
import Lecture2;

public class TestLecture2 {

    public static void main(String[] args) {
        int x = Lecture2.max2(5, -3);
    }
}
```

From C to Java

CMPE 212

As in C, indentation is used only for readability

- ▶ the compiler does not care if your code is indented
 - ▶ but use the examples in the slides and course notes for guidance on good Java programming style

```
// this compiles cleanly, but don't format your code  
// like this
```

```
public class Lecture2 { public static int  
max2(int a, int b) {  
int twiceMax = a + b + Math.abs(a - b);  
    return twiceMax / 2;}  
}
```

Java uses braces to delimit the bodies of classes, methods, if statements, and loops

- ▶ the braces are required

```
public class Lecture2 {
```

```
    public static int max2(int a, int b) {
        int twiceMax = a + b + Math.abs(a - b);
        return twiceMax / 2;
    } // end of method body
```

```
} // end of class body
```

For the time being, all of our classes will be public

- ▶ the **public** access modifier on a top-level class means that the class is visible to all other classes

```
public class Lecture2 {
```

```
    public static int max2(int a, int b) {  
        int twiceMax = a + b + Math.abs(a - b);  
        return twiceMax / 2;  
    }
```

```
}
```

For the time being, all of our methods will be public and static

- ▶ the **public** access modifier on a method means that the method is callable from within any class

```
public class Lecture2 {  
  
    public static int max2(int a, int b) {  
        int twiceMax = a + b + Math.abs(a - b);  
        return twiceMax / 2;  
    }  
}
```

For the time being, all of our methods will be public and static

- ▶ the **static** modifier on a method means that the method is associated with its class
- ▶ which is why you use the class name in addition to the method name when calling a static method

```
public class Lecture2 {  
  
    public static int max2(int a, int b) {  
        int twiceMax = a + b + Math.abs(a - b);  
        return twiceMax / 2;  
    }  
}
```

As in C functions, the parameters of a Java method must have types

- e.g., **max2** has two parameters of type **int**

```
public class Lecture2 {  
  
    public static int max2(int a, int b) {  
        int twiceMax = a + b + Math.abs(a - b);  
        return twiceMax / 2;  
    }  
}
```

As in C functions, a Java method is allowed to return zero or one value

- ▶ if a method returns a value, then it must declare the type of the value that it returns
- ▶ the type appears immediately before the method name
 - ▶ e.g., `max2` returns an `int` value

```
public class Lecture2 {  
  
    public static int max2(int a, int b) {  
        int twiceMax = a + b + Math.abs(a - b);  
        return twiceMax / 2;  
    }  
}
```

A Java method is allowed to return zero or one value

- ▶ if a method does not return a value, then it must declare that it returns the type **void**
- ▶ e.g., a main always returns **void** in Java

```
import Lecture2;

public class TestLecture2 {

    public static void main(String[] args) {
        int x = Lecture2.max2(5, -3);
    }
}
```

There must be a **return** statement in a method that returns a value

- ▶ the method stops running immediately after the return statement finishes running

```
public class Lecture2 {  
  
    public static int max2(int a, int b) {  
        int twiceMax = a + b + Math.abs(a - b);  
        return twiceMax / 2;  
    }  
}
```

type must be compatible with
the declared return type of the method

A **void** method has no return statement

```
import Lecture2;

public class TestLecture2 {

    public static void main(String[] args) {
        int x = Lecture2.max2(5, -3);
        // no return statement
    }
}
```

A **void** method has no return statement

- ▶ but an empty **return** statement is legal

```
import Lecture2;

public class TestLecture2 {

    public static void main(String[] args) {
        int x = Lecture2.max2(5, -3);
        return;
    }
}
```

if statements are similar in C and Java:

```
public static double max3(double a, double b, double c) {  
    double result;  
    if (a >= b && a >= c) {  
        result = a;  
    }  
    else if (b >= a && b >= c) {  
        result = b;  
    }  
    else {  
        result = c;  
    }  
    return result;  
}
```

We can remove the **else** clause by assuming the maximum value is **c**:

```
public static double max3(double a, double b, double c) {  
    double result = c;  
    if (a >= b && a >= c) {  
        result = a;  
    }  
    else if (b >= a && b >= c) {  
        result = b;  
    }  
    return result;  
}
```

The conditions must appear inside parentheses

```
public static double max3(double a, double b, double c) {  
    double result = c;  
    if (a >= b && a >= c) {  
        result = a;  
    }  
    else if (b >= a && b >= c) {  
        result = b;  
    }  
    return result;  
}
```

The *scope* of a variable is the region of code where a declared variable name is usable

- ▶ the scope starts where the variable was declared and goes to the end of the block that the variable was declared in

parameters: scope = method body



```
public static double max3(double a, double b, double c) {  
    double result = c;  
    if (a >= b && a >= c) {  
        result = a;  
    }  
    else if (b >= a && b >= c) {  
        result = b;  
    }  
    return result;  
}
```

The *scope* of a variable is the region of code where a declared variable name is usable

- ▶ in Java, the scope starts where the variable was declared and goes to the end of the block that the variable was declared in

```
public static double max3(double a, double b, double c) {  
    double result = c;                      result: scope = from here to end of method body  
    if (a >= b && a >= c) {  
        result = a;  
    }  
    else if (b >= a && b >= c) {  
        result = b;  
    }  
    return result;  
}
```

The *scope* of a variable is the region of code where a declared variable name is usable

- ▶ in Java, the scope starts where the variable was declared and goes to the end of the block that the variable was declared in

```
public static double max3(double a, double b, double c) {  
    if (a >= b && a >= c) {  
        double result = a;          result: scope = just this line!  
    }  
    else if (b >= a && b >= c) {  
        double result = b;          result: scope = just this line!  
    }  
    return result;  
}
```

compile-time error:
no declared variable named **result**

A method may have more than one return statement

- ▶ but there cannot be reachable code after a return statement
- ▶ some consider multiple return statements to be bad form

```
public static double max3(double a, double b, double c) {  
    if (a >= b && a >= c) {  
        return a;  
        // no code allowed here  
    }  
    else if (b >= a && b >= c) {  
        return b;  
        // no code allowed here  
    }  
    return c;  
    // no code allowed here  
}
```

A method that returns a value must always return a value

- most compilers do not attempt logical deduction

```
public static double max3(double a, double b, double c) {  
    if (a >= b && a >= c) {  
        return a;  
    }  
    else if (b >= a && b >= c) {  
        return b;  
    }  
    else if (c >= a && c >= b) {  
        return c;  
    }  
    // compile-time error: compiler does not recognize that the method  
    // always returns a value  
}
```

in this example, the compiler sees three if statements and assumes that it is possible that none of the conditions are true which leads to program flow reaching the end of the method without executing a return statement

&& is the logical AND operator

```
public static double max3(double a, double b, double c) {  
    double result = c;  
    if (a >= b && a >= c) {  
        result = a;  
    }  
    else if (b >= a && b >= c) {  
        result = b;  
    }  
    return result;  
}
```

`||` is the logical OR operator

- ▶ `&&` has higher precedence than `||`

```
public static boolean isLeapYear (int year) {  
    if (year % 400 == 0 || year % 4 == 0 &&  
        year % 100 != 0) {  
        return true;  
    }  
    return false;  
}
```

Use parentheses if you want to be explicit about precedence

```
public static boolean isLeapYear (int year) {  
    if (year % 400 == 0 || (year % 4 == 0 &&  
                            year % 100 != 0)) {  
        return true;  
    }  
    return false;  
}
```

Start training yourself to avoid writing an if statement when one is not required:

```
public static boolean isLeapYear (int year) {  
    return year % 400 == 0 ||  
        (year % 4 == 0 && year % 100 != 0);  
}
```

! is the logical NOT operator

- ▶ assume **isLeapYear** is defined in the class **Lecture3**

```
public static int daysInFeb(int year) {  
    if (!Lecture3.isLeapYear(year)) {  
        return 28;  
    }  
    return 29;  
}
```

From C to Java: Arrays

CMPE 212

Arrays

- ▶ arrays have many similarities in C and Java, but Java arrays are easier and safer to use
- ▶ arrays are somewhat uncommon in Java because the Java Standard Library provides more flexible container objects such as lists, sets, and maps
- ▶ in Java, an array is a container object that holds a *fixed number* of values of a *single type*
- ▶ the length of an array is established when the array is created

Arrays

- ▶ to declare an array, you use the element type followed by an empty pair of square brackets

```
double[] collection;  
// collection is an array of double values
```

As in C, an array can initialize its size and elements by writing the elements in a comma separated list

- ▶ but this can only be done when the array variable is declared

```
char[] t = {'a', 'b', 'c', 'd'}; // ok
```

```
char[] u;  
u = {'x', 'y', 'z'}; // oops, syntax error  
// cannot use {} to assign elements  
// to an array
```

Arrays are normally created using the **new** operator and specifying the *capacity* or *length* of the array

- ▶ the capacity is the maximum number of elements that the array can hold
- ▶ capacity is specified in square brackets

```
double[] collection;
```

```
// collection is an array of double values
```

```
collection = new double[10];
```

```
// collection is an array of 10 double values
```

Arrays of primitive type elements have all the elements initialized to a default value when using **new**

```
import java.util.Arrays;

public class Lecture4 {

    public static void main(String[] args) {
        boolean[] someBools = new boolean[3];
        int[] someInts = new int[4];
        double[] someDbls = new double[5];
        System.out.println(Arrays.toString(someBools));
        System.out.println(Arrays.toString(someInts));
        System.out.println(Arrays.toString(someDbls));
    }
}
```

Default values for arrays

Array type	Default element value
<code>boolean[]</code>	<code>false</code>
<code>byte[]</code>	<code>0</code>
<code>char[]</code>	<code>0</code>
<code>short[]</code>	<code>0</code>
<code>int[]</code>	<code>0</code>
<code>long[]</code>	<code>0L</code>
<code>float[]</code>	<code>0.0F</code>
<code>double[]</code>	<code>0.0</code>

To fill an existing array so that all elements have the same value, use the method **Arrays.fill**

```
import java.util.Arrays;

public class Lecture4 {
    public static void main(String[] args) {
        double[] someDbls = new double[10];
        System.out.println(Arrays.toString(someDbls));

        // fill array with 1.0s
        Arrays.fill(someDbls, 1.0);
        System.out.println(Arrays.toString(someDbls));
    }
}
```

Arrays

- ▶ the number of elements in the array is stored in the public field named **length**

```
double[] collection;
```

```
// collection is an array of double values
```

```
collection = new double[10];
```

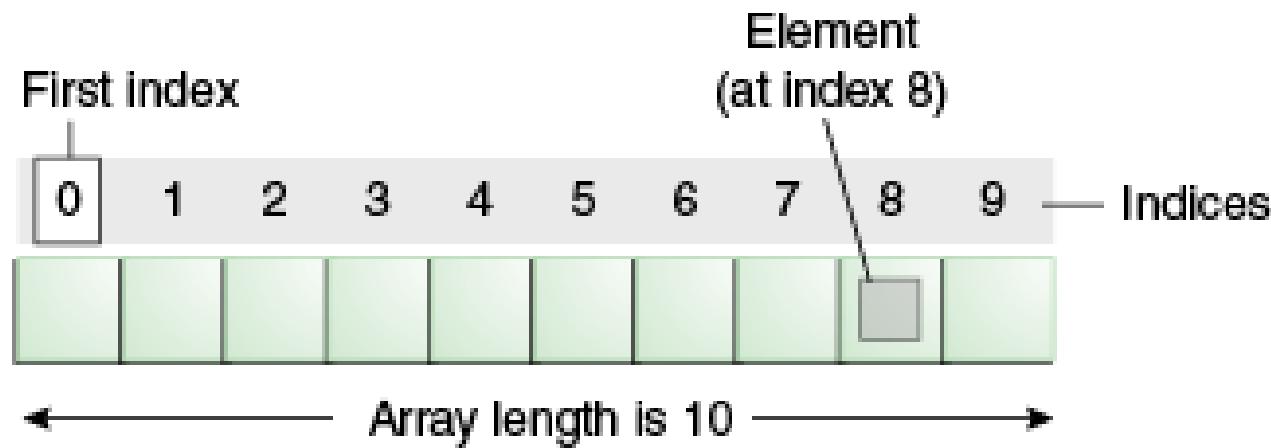
```
// collection is an array of 10 double values
```

```
int n = collection.length;
```

```
// the public field length holds the number of elements
```

Arrays

- ▶ the values in an array are called elements
- ▶ the elements can be accessed using a zero-based index



- ▶ the last valid index for an array **a** is (**a.length - 1**)

Arrays

- ▶ the elements can be accessed using a zero-based index
(similar to strings)

```
collection[0] = 100.0;           // set an element
double x = collection[0];       // get an element
collection[1] = 100.0;
collection[2] = 100.0;
collection[3] = 100.0;
collection[4] = 100.0;
collection[5] = 100.0;
collection[6] = 100.0;
collection[7] = 100.0;
collection[8] = 100.0;
collection[9] = 100.0;           // collection[9] is last element
collection[10] = 100.0;          // ArrayIndexOutOfBoundsException
```

Arrays

- ▶ unlike in C, attempting to access an element using an invalid index immediately causes an error
- ▶ errors in Java are typically indicated by an exception being thrown
 - ▶ an exception is a special kind of object that is created when an exceptional event occurs
 - ▶ the exception object may contain information regarding the event
- ▶ when an exception occurs, the program suspends running at the point where the exception occurred
 - ▶ the Java runtime system then looks for code called an exception handler (more on this later in the course)
 - if an exception handler is not found, then the program stops running

A common operation when working with C arrays is to loop over the elements of the array

- e.g., to manually find the maximum value in a array

```
double max_elem(const double arr[], size_t size) {
    if (size == 0) {
        // error... now what?
    }
    double hi = arr[0];
    for (size_t i = 0; i < size; i++) {
        if (arr[i] > hi) {
            hi = arr[i];
        }
    }
    return hi;
}
```

Java supports C-style for loops

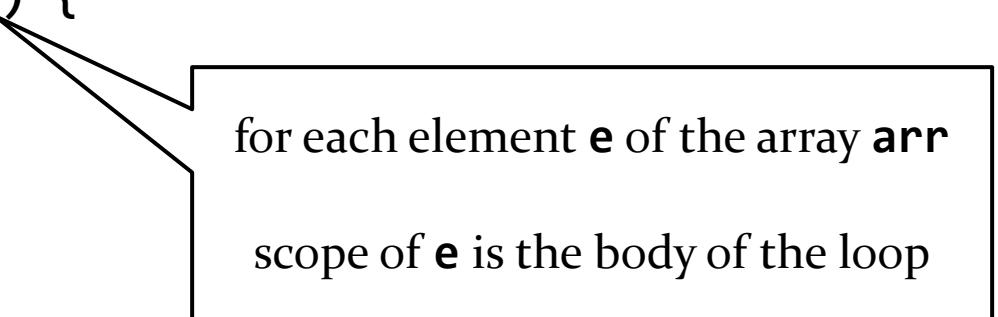
- ▶ was the only kind of for loop until Java 5

```
public static double maxElem(double[] arr) {  
    if (arr.length == 0) {  
        throw new IllegalArgumentException("array length = 0");  
    }  
    double hi = arr[0];  
    for (int i = 0; i < arr.length; i++) {  
        if (arr[i] > hi) {  
            hi = arr[i];  
        }  
    }  
    return hi;  
}
```

scope of **i** is the body of the loop

Java also has a for-each loop that lets the programmer do something for each element of the array without specifying an index.

```
public static double maxElem(double[] arr) {  
    if (arr.length == 0) {  
        throw new IllegalArgumentException("array length = 0");  
    }  
    double hi = arr[0];  
    for (double e : arr) {  
        if (e > hi) {  
            hi = e;  
        }  
    }  
    return hi;  
}
```



for each element **e** of the array **arr**
scope of **e** is the body of the loop

for loops

- ▶ a regular for loop has four main parts
 1. an initialization expression
 2. a termination condition
 3. an update expression
 4. a loop body
 - ▶ parts 1, 2, and 3 are technically all optional but it is a little unusual to see a for loop that does not have all four parts
 - ▶ the two semi-colons separating parts 1 and 2 and 2 and 3 are always required

The initialization expression initializes the loop

- ▶ executed exactly once when the loop begins
- ▶ usually, a loop variable is declared and initialized in the initialization expression

```
for (int i = 0; i < arr.length; i++) {  
}
```

The termination expression must evaluate to **true** for the loop to run the next iteration

- ▶ executed *before* every loop iteration
- ▶ if it evaluates to **false** then the loop stops iterating

```
for (int i = 0; i < arr.length; i++) {  
}
```

The update expression is evaluated after each iteration of the loop

- ▶ almost always modifies the value of the loop variable

```
for (int i = 0; i < arr.length; i++) {  
}
```

If the loop body must access more than one element of an array in each iteration, then you should probably use a regular for loop instead of a for-each loop

- e.g., test if an array is sorted in ascending order

```
public static boolean isSorted(int[] arr) {  
    boolean isSorted = true;  
    for (int i = 1; i < arr.length; i++) {  
        if (arr[i - 1] > arr[i]) {  
            isSorted = false;  
        }  
    }  
    return isSorted;  
}
```

iteration 1

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i-1 i



iteration 2

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i-1 **i**



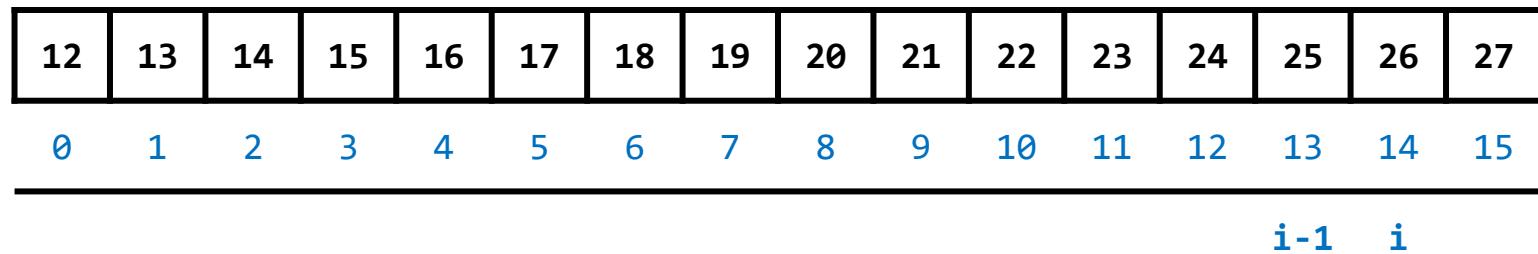
iteration 3

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

$i-1$ i



iteration 14



iteration 15

12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i-1 **i**



The **break** statement will immediately break out of the loop causing program flow to proceed to the statement immediately after the loop.

```
public static boolean isSorted(int[] arr) {  
    boolean isSorted = true;  
    for (int i = 1; i < arr.length; i++) {  
        if (arr[i - 1] > arr[i]) {  
            isSorted = false;  
            break; // not sorted, no need to continue iterating  
        }  
    }  
    return isSorted;  
}
```

iteration 3, **break** statement runs stopping the loop

12	13	99	15	16	17	18	19	20	21	22	23	24	25	26	27	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i-1 **i**



If the loop body must access more than one element of an array in each iteration, then you should probably use a regular for loop instead of a for-each loop

- e.g., reverse the order of the elements in an array

```
public static void reverse(int[] a) {  
    int j = a.length - 1;  
    for (int i = 0; i < j; i++) {  
        // swap a[i] and a[j]  
        int tmp = a[i];  
        a[i] = a[j];  
        a[j] = tmp;  
        j--;  
    }  
}
```

iteration 1

iteration 2

iteration 3

A horizontal array of 16 boxes, each containing a number from 0 to 15. The numbers 14, 25, and 15 are highlighted in blue. Below the array, the word "index" is followed by a row of numbers from 0 to 15. A vertical line connects index 0 to the number 14, and another vertical line connects index 15 to the number 15. The label "i" is placed below index 0, and the label "j" is placed below index 15.

27	26	14	15	16	17	18	19	20	21	22	23	24	25	13	12	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i

j

iteration 8

27	26	25	24	23	22	21	19	20	18	17	16	15	14	13	12	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i j



iteration 8 ends

27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

i j



In the previous example, there are two loop variables of *the same type*

- ▶ it is possible to incorporate both loop variables into the initialization and update expressions

```
public static void reverse(int[] a) {  
    for (int i = 0, j = a.length - 1; i < j; i++, j--) {  
        // swap a[i] and a[j]  
        int tmp = a[i];  
        a[i] = a[j];  
        a[j] = tmp;  
    }  
}
```

If the loop body must access elements from two or more arrays, then you must use a regular for loop

- ▶ e.g., return a new array by summing corresponding elements of two input arrays

```
public static int[] sum(int[] a, int[] b) {  
    if (a.length != b.length) {  
        throw new IllegalArgumentException("array lengths differ");  
    }  
    int[] c = new int[a.length];  
    for (int i = 0; i < a.length; i++) {  
        c[i] = a[i] + b[i];  
    }  
    return c;  
}
```

In Python, you can slice a list to get the elements located in a sublist of the original list:

```
t = [10, 11, 12, 13, 14, 15]
slc = t[2:5]

# slc is the sublist [t[2], t[3], t[4]]
```

Java has no built-in slice operator, but you can write your own method to slice an array:

```
public static int[] slice(int[] arr, int start, int stop) {  
    if (start > stop) {  
        throw new IllegalArgumentException("start > stop");  
    }  
    // what other input errors do we need to check for?  
  
    int[] slc = new int[stop - start];  
    // copy elements from arr into slc  
    for (int i = start; i < stop; i++) {  
        slc[i - start] = arr[i];  
    }  
    return slc;  
}
```

Calling the Java version of `slice`:

```
public class Lecture4 {  
    public static int[] slice(int[] arr, int start, int stop) {  
        // see previous slide  
    }  
  
    public static void main(String[] args) {  
        int[] t = {10, 11, 12, 13, 14, 15};  
        int[] slc = Lecture4.slice(t, 2, 5);  
    }  
}
```

In Python, you can (shallow copy) a list by slicing the entire list:

```
t = [10, 11, 12, 13, 14, 15]  
copy = t[:]
```

In Java, you could use **slice**, but arrays have a **clone()** method that you can use instead:

```
public class Lecture4 {  
    public static int[] slice(int[] arr, int start, int stop) {  
        // see previous slide  
    }  
  
    public static void main(String[] args) {  
        int[] t = {10, 11, 12, 13, 14, 15};  
        int[] slc = Lecture4.slice(t, 2, 5);  
        int[] copy = t.clone();  
    }  
}
```

char and String

A string literal is a sequence of characters between double quotes

- ▶ the characters making up a string have the primitive type **char**

```
String s = "rub-a-dub-dub";
char c = s.charAt(0);           // get first character of s
```

char literals are delimited by single quotes

- ▶ there must be exactly one character inside the quotes
 - ▶ no such thing as an empty **char**

```
char c = 'a';  
char blank = ' ';  
char oops = '';      // does not compile, no such thing  
                     // as an empty char
```

The type **char** is actually an integer type.

- ▶ a **char** value is an integer that is mapped to a character
 - ▶ the mapping is defined by the Unicode Standard, version 6.0.0 (unlike C)

```
System.out.println((int) Character.MIN_VALUE);      // 0
System.out.println((int) Character.MAX_VALUE);      // 65_535

char c = 65;
System.out.println(c);                            // 'A'
c += 25;                                         // add 25 to c
System.out.println(c);                            // 'Z'
```

You can do some arithmetic with **char**, but the results are unintuitive, especially when **int** values are involved.

```
char c = 'A';
c++;           // ok, add 1 to c
c += 1;         // ok, add 1 to c
c -= 1;         // ok, subtract 1 from c
c = c + 1;      // error, cannot convert from int to char

char d = 'B';
char e = c + d; // error, cannot convert from int to char
```

Arithmetic with **char** is useful when you want to generate a sequence of characters.

- e.g., generate all pairs of two letter upper-case strings

```
for (char c = 'A'; c <= 'Z'; c++) {  
    for (char d = 'A'; d <= 'Z'; d++) {  
        String s = "" + c + d;  
        System.out.println(s);  
    }  
}
```

string concatenation,
not **char** arithmetic

Types, classes, and objects

- ▶ a *type* is a set of values and the operations that can be done with those values
- ▶ a *class* defines a reference type in Java
- ▶ an *object* is an instance of a class

The **String** class

- ▶ a **String** object represents text (a sequence of characters)
 - ▶ very widely used in Java programs
- ▶ because they are so widely used, the Java language lets a client perform actions with **String** objects that cannot be performed with other types of objects
 - ▶ e.g., string literals exist, but no literals exist for any other reference type
 - ▶ e.g., the + operator is defined for strings, but no arithmetic or comparison operators are defined for any other reference type

The **String** class

- ▶ documentation for the String class can be found at:
 - ▶ <https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>
- ▶ **String** objects are immutable
 - ▶ the sequence of characters in a **String** object can never be changed
- ▶ methods that seem like they change the characters of a string actually return a reference to a new **String** object

The **toUpperCase** and **toLowerCase** methods return uppercase and lowercase versions of a string without modifying the original string

```
String s = "hello";
String up = s.toUpperCase();
System.out.println("s : " + s);
System.out.println("up : " + up);

// common error
s.toUpperCase();      // try to make "hello" be "HELLO"?
```

You require an object reference to call a non-static method.

```
String s = "hello";
String up = s.toUpperCase();    // use the object referred
                                // to by s to call a
                                // non-static method
```

String does have static methods as well (see the documentation).

- ▶ use the class name to call a static method
 - ▶ e.g., to convert the value of a **double** to its **String** representation

```
String s = String.valueOf(1.0);
```

```
// alternatively  
String t = "" + 1.0;
```

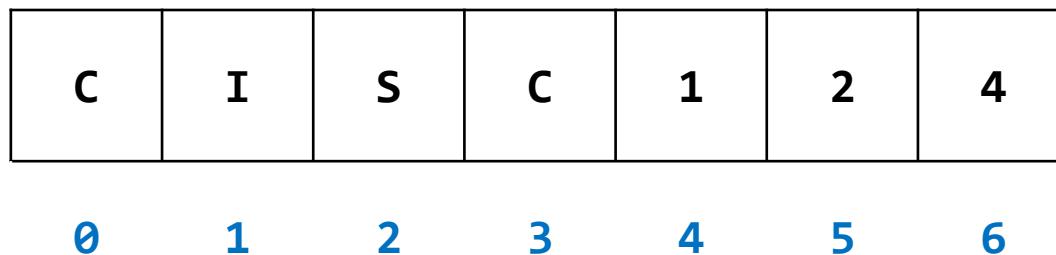
Use the **equals** method to test if two strings are equal
(represent identical sequences of characters)

- ▶ do not use `==` or `!=` to compare two strings for equality

```
String s = "hello";
String t = new String("hello");
boolean boo = s.equals(t);           // true
boo = s == t;                      // false!
```

Like in C, a string is a numbered sequence

- ▶ each **char** in the sequence has an integer index starting from zero



The number of characters in a string is called its length

- returned by the method **length**
- Java strings are *not null terminated*

```
String courseName = "CISC124";  
int len = courseName.length();      // 7
```

```
String empty = "";  
len = empty.length();                // 0
```

The method **charAt** returns the character at a specified index

- ▶ index must be between **0** and **length()** - **1**, inclusive

```
String courseName = "CISC124";
for (int i = 0; i < courseName.length(); i++) {
    char c = courseName.charAt(i);
    System.out.println(c);
}
```

Here is a method that counts the number of times a specified character appears in a string:

```
public class Strings {  
    public static int freq(String s, char target) {  
        int count = 0;  
        for (int i = 0; i < s.length(); i++) {  
            if (s.charAt(i) == target) {  
                count++;  
            }  
        }  
        return count;  
    }  
}
```

Here is an example of using the `freq` method:

```
public class TestStrings {  
    public static void main(String[] args) {  
        String courseName = "CMPE212";  
        int count = Strings.freq(courseName, 'C');  
        System.out.println(count); // 2  
  
        count = Strings.freq(courseName, 'c');  
        System.out.println(count); // 0  
  
        count = Strings.freq(courseName, '2');  
        System.out.println(count); // 2  
    }  
}
```

The **contains** method returns **true** if a string contains a specified substring

```
String courseName = "CMPE212";
boolean boo = courseName.contains("CMPE");      // true
boo = courseName.contains("cmpe");               // false
boo = courseName.contains("1");                  // true

// search for a char?
boo = courseName.contains('C');                 // error
boo = courseName.contains("") + 'C');            // true
```

The method **indexOf** returns the first location of a specified character in a string

- ▶ returns **-1** if the character is not in the string

```
String courseName = "CMPE212";
int index = courseName.indexOf('C');
System.out.println(index); // 0
```

```
index = courseName.indexOf('P');
System.out.println(index); // 2
```

```
index = courseName.indexOf('2');
System.out.println(index); // 4
```

```
index = courseName.indexOf('z');
System.out.println(index); // -1
```

The method **lastIndexOf** returns the last location of a specified character in a string

- ▶ returns **-1** if the character is not in the string

```
String courseName = "CMPE212";
int index = courseName.lastIndexOf('C');
System.out.println(index); // 0

index = courseName.lastIndexOf('P');
System.out.println(index); // 2

index = courseName.lastIndexOf('2');
System.out.println(index); // 6

index = courseName.lastIndexOf('z');
System.out.println(index); // -1
```

The **equals** method tests if a string is equal to a specified string

- ▶ *do not* use `==` to compare two strings for equality

```
String cmpe = "CMPE212";
String math = "MATH212";
boolean eq = cmpe.equals(math);      // false
```

```
// case matters
eq = cmpe.equals("cmpe212");        // false
```

The **+** operator is the concatenation operation

- joins the characters of two strings to create a new string

```
String name1 = "James Bond 007";
String name2 = "James " + "Bond " + "007";
boolean eq = name1.equals(name2); // true
```

Primitive values can also be concatenated onto a string

```
String name1 = "James Bond 007";
String name2 = "James " + "Bond " + "007";
boolean eq = name1.equals(name2); // true
```

```
String name3 = "James Bond " + 0 + 0 + 7;
eq = name1.equals(name3); // true
```

But beware of the order of the operands:

```
String laugh = 'h' + 'a' + "ha";      // 201ha
```

The expression '**h**' + '**a**' + "**ha**" involves operators all having the same precedence, so it is evaluated from left to right:

1. '**h**' + '**a**' is the sum of two **char** literals; addition is not defined for **char** so both operands are promoted to **int** and then summed to yield **201**
2. **201** + "**ha**" involves a string so string concatenation is performed to yield "**201ha**"

The **substring** methods allow the programmer to get a copy of a contiguous part of a string

- indexes are tested and an exception is thrown if an index is invalid

```
String s = "abcdefg";
String t = s.substring(1);           // "bcdefg"
String u = s.substring(3, 4);        // "d"
String v = s.substring(5, 7);        // "fg"
```

Use the **compareTo** method to compare two strings by lexicographical order

- ▶ similar to **strcmp** in C
- ▶ for two string references **s** and **t**

s.compareTo(t) returns an integer less than zero

if **s** precedes **t**
lexicographically

s.compareTo(t) returns zero

if **s.equals(t)** is
true

s.compareTo(t) returns an integer greater than
zero

if **s** follows **t**
lexicographically

```
String s = "aardvark";
```

```
String t = "zebra";
```

Expression	Return value
s.compareTo(t)	less than zero
s.compareTo(s)	zero
t.compareTo(s)	greater than zero

The **replace** methods will replace all occurrences of a **char** or substring with a specified **char** or substring

```
String s = "sparring with a purple porpoise";
String t = s.replace('p', 't');
// starring with turtle tortoise

s = "hiho, hiho, it's off to work we go";
t = s.replace("hiho", "ohno");
// ohno, ohno, it's off to work we go
```

Many other methods

- ▶ <https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

Floating-point

The types **float** and **double** approximate real numbers in Java.

- ▶ because they occupy a fixed, finite amount of memory, they cannot represent every real number exactly

```
double x = 0.1 + 0.1 + 0.1;  
boolean eq = x == 0.3;           // false!
```

```
import java.math.BigDecimal;

public class FloatingPoint {

    public static String allDigits(double val) {
        BigDecimal big = new BigDecimal(val);
        return big.toString();
    }

    public static void main(String[] args) {
        double x = 0.1 + 0.1 + 0.1;
        boolean eq = x == 0.3;
        System.out.println(eq);

        System.out.println(allDigits(0.1));
        System.out.println(allDigits(0.1 + 0.1));
        System.out.println(allDigits(0.1 + 0.1 + 0.1));
        System.out.println(allDigits(0.1 + 0.1 + 0.1 + 0.1));
        System.out.println(allDigits(0.1 + 0.1 + 0.1 + 0.1 + 0.1));

        System.out.println(allDigits(0.3));
    }
}
```

Aside

- ▶ the output from the previous program is somewhat misleading
- ▶ a **double** has between 15 and 17 significant digits when expressed in base-10 (decimal)
 - ▶ but the program prints out much more than 17 digits
- ▶ what is happening is that the literal **0.1** is converted to the nearest **double** value that can be represented
 - ▶ this value is in base-2 (binary)
- ▶ the program is printing out the result of converting the base-2 value back to base-10

IEEE 754

- ▶ the IEEE Standard for Floating-Point Arithmetic (IEEE 754) is the standard for floating-point arithmetic used in most modern CPUs intended for general purpose computing
- ▶ the details of the standard are beyond the scope of this course, but you should know that every floating-point number has the form

$$\pm s \times 2^e$$

where s (called the significand) and e (the exponent) are both integer values

Why is 0.1 actually 0.1000...

- ▶ **0.1** cannot be represented exactly in floating-point

Proof by contradiction: Assume that we can find integer values s and e such that

$$s \times 2^e = 0.1$$

(note that e must be negative because s is an integer value and the right-hand side of the equality is less than 1)

Why is 0.1 actually 0.1000...

Then,

$$s \times 2^e = 0.1$$

$$10s \times 2^e = 1$$

$$10s = \frac{1}{2^e}$$

$$10s = 2^{-e}$$

$$2 \times 5 \times s = 2 \times 2 \times 2 \times \cdots \times 2$$

which is a contradiction because the prime factorization of the left-hand side includes at least one 5 but the prime factorization of the right-hand side includes only 2's.

IEEE 754

- ▶ **float** and **double** are 32 bit and 64 bit binary values
- ▶ trying to study the behavior of floating-point arithmetic using their true representation is awkward
 - ▶ large number digits
 - ▶ humans are used to decimal values rather than binary values
- ▶ consider two different decimal representations using only four digits

Four digit floating-point

$$\pm \underbrace{d_1 \ d_2 \ d_3}_{\text{significand } s} \times 10^{\underbrace{-5}_{\text{exponent } e}}$$

$d_1 \neq 0$

$-5 \leq e \leq 4$

d_4 **-5**

Four digit floating-point

Four digit floating-point	Decimal value	Difference
100×10^{-5}	0.00100	
101×10^{-5}	0.00101	0.00001
100×10^{-4}	0.0100	
101×10^{-4}	0.0101	0.0001
100×10^2	10000.	
101×10^2	10100.	100.
100×10^4	100000.	
101×10^4	101000.	10000.

Unit in the last place (ulp)

- ▶ notice that the distance between adjacent floating-point values changes depending on the value of the exponent
- ▶ this distance is called an *ulp*
 - ▶ useful for measuring error in floating-point values
- ▶ the methods **ulp(double d)** and **ulp(float f)** in the class **java.lang.Math** return the value of an ulp for a specified floating-point value

Error

- ▶ absolute error is defined as

$$\text{absolute error} = |\hat{x} - x|$$

where \hat{x} is the true value of a quantity and x is the approximate value of the quantity

- ▶ converting a real value that fits in the range of a floating-point type to its nearest floating-point value has an error of $\frac{1}{2}\text{ulp}$

Error

- ▶ to compute an error in ulps
 1. write \hat{x} as a floating-point value (significand and exponent)
 2. write x as a floating-point value but using the same exponent as \hat{x}
 3. compute $|\hat{x} - x|$ using the values computed in Steps 1 and 2

Example

- ▶ true value $\hat{x} = 1.29, x = 1.25$

$$\begin{aligned}\text{absolute error} &= |\hat{x} - x| \\ &= |129 \times 10^{-2} - 125 \times 10^{-2}| \\ &= 4 \times 10^{-2} \\ &= 4 \text{ ulps}\end{aligned}$$

Example

- ▶ true value $\hat{x} = 12.1, x = 0.5$

$$\begin{aligned}\text{absolute error} &= |\hat{x} - x| \\ &= |121 \times 10^{-1} - 5 \times 10^{-1}| \\ &= 116 \times 10^{-1} \\ &= 116 \text{ ulps}\end{aligned}$$

Example

- ▶ sometimes the true value is a real number as opposed to a floating-point value
- ▶ in such cases, write the true value as a floating-point value keeping all values after the decimal point
- ▶ true value $\hat{x} = 59287.5603$, $x = 58900$

$$\begin{aligned}\text{absolute error} &= |\hat{x} - x| \\ &= |592.875603 \times 10^2 - 589 \times 10^2| \\ &= 3.875603 \times 10^2 \\ &= 3.875603 \text{ ulps}\end{aligned}$$

Relative error

- ▶ relative error is simply the absolute error divided by the true value

$$\text{relative error} = \left| \frac{\hat{x} - x}{\hat{x}} \right|$$

- ▶ use real arithmetic to compute relative error

Addition/subtraction

- ▶ if two floating-point operands have the same exponent, then addition/subtraction can be performed simply by adding/subtracting the significands
- ▶ e.g., $12.1 + 63.8$

$$x = 121 \times 10^{-1}$$

$$y = 638 \times 10^{-1}$$

$$x + y = 759 \times 10^{-1}$$

e.g., $81500 + 13600$

$$x = 815 \times 10^2$$

$$y = 136 \times 10^2$$

$$x + y = 951 \times 10^2$$

Addition/subtraction

- ▶ if the result produces an extra digit, then write the result using the extra digit and round to remove a digit
- ▶ e.g., $82.1 + 63.8$

$$x = 821 \times 10^{-1}$$

$$y = 638 \times 10^{-1}$$

$$\begin{aligned}x + y &= 1459 \times 10^{-1} \text{ intermediate result} \\&= 146 \times 10^0\end{aligned}$$

e.g., $81500 + 93600$

$$x = 815 \times 10^2$$

$$y = 936 \times 10^2$$

$$\begin{aligned}x + y &= 1751 \times 10^2 \text{ intermediate result} \\&= 175 \times 10^3\end{aligned}$$

Addition/subtraction

- ▶ if the operands have different exponents, then the operand with the smaller exponent has its significand scaled so that the exponents are equal
- ▶ e.g., $4,320,000 + 12.1$

$$x = 432 \times 10^4$$

$$y = 000.00121 \times 10^4$$

$$\begin{aligned}x + y &= 432.00121 \times 10^4 \text{ intermediate result} \\&= 432 \times 10^4\end{aligned}$$

- ▶ observe that a total of 8 digits are required to compute the result

Addition/subtraction

- ▶ using extra digits was not always feasible in computer hardware (but it is commonly done on modern CPUs)
- ▶ instead of using extra digits, suppose that we discard the extra digits from the operand with the smaller exponent
- ▶ e.g., $4,320,000 + 12.1$

$$\begin{aligned}x &= 432 \times 10^4 \\y &= 000 \times 10^4 \quad \text{discard extra digits} \\x + y &= 432 \times 10^4\end{aligned}$$

which produces the same result as the previous slide

Addition/subtraction

- ▶ without using extra digits, addition/subtraction can produce large errors
- ▶ e.g., $10.5 - 9.98$

$$\begin{aligned}x &= 105 \times 10^{-1} \\y &= 099 \times 10^{-1} \quad \text{discard extra digits} \\x - y &= 006 \times 10^{-1} \\&= 600 \times 10^{-3}\end{aligned}$$

- ▶ but the true result is 520×10^{-3} which is an error of 80 ulps
 - ▶ IEEE754 says that the error must be within 0.5 ulps for a basic arithmetic operation

Guard digits

- ▶ suppose that we use one extra digit to the right of the decimal point (called a *guard digit*)
- ▶ e.g., $10.5 - 9.98$

$$x = 105.0 \times 10^{-1}$$

$$y = 099.8 \times 10^{-1} \quad \text{discard extra digits}$$

$$x - y = 005.2 \times 10^{-1}$$

$$= 520 \times 10^{-3} \quad \text{normalize and round}$$

which matches the true result!

Guard digits

- ▶ a single guard digit is not sufficient to ensure 0.5 ulp error (but two guard digits suffice)
- ▶ e.g., $110 - 8.59$

$$x = 110.0 \times 10^0$$

$$y = 008.5 \times 10^0 \quad \text{discard extra digits}$$

$$x - y = 101.5 \times 10^0$$

$$= 102 \times 10^0 \quad \text{normalize and round}$$

compared to the true value of 101.41 (0.59 ulp error)

Cancellation

- ▶ subtracting two similar floating-point values results in a phenomenon called *cancellation* or *loss of significance*
- ▶ consider subtracting two similar 10-digit floating-point values:

$$x = 1234567890 \times 10^0$$

$$y = 1234567889 \times 10^0$$

$$x - y = 0000000001 \times 10^0$$

$$= 1000000000 \times 10^{-9} \quad \text{normalize}$$

- ▶ the difference has only one significant digit even though both operands have 10 significant digits

Cancellation

- ▶ called cancellation because the leftmost digits (or the *high order* digits, or the *most significant* digits) of the two operands cancel one another out leaving only the rightmost digits (or the *low order* digits, or the *least significant* digits)
- ▶ if the two operands contain no error then the phenomenon is called *benign* cancellation

Catastrophic cancellation

- ▶ catastrophic cancellation occurs when one or both of the operands contain error
 - ▶ for example, when the operands are computed using floating-point arithmetic
- ▶ consider computing the value of $b^2 - 4ac$ when $b = 3.34, a = 1.22, c = 2.28$
- ▶ exact value:

$$\begin{aligned}b^2 - 4ac &= 3.34^2 - 4(1.22)(2.28) \\&= 11.1556 - 11.1264 \\&= 0.0292 \\&= 292 \times 10^{-4}\end{aligned}$$

Catastrophic cancellation

- ▶ consider computing the value of $b^2 - 4ac$ using floating-point arithmetic with a three digit significand:

$$b^2 = 111.556 \times 10^{-1}$$

$$= 112 \times 10^{-1} \quad \text{rounded}$$

$$4ac = 111.264 \times 10^{-1}$$

$$= 111 \times 10^{-1} \quad \text{rounded}$$

$$b^2 - 4ac = 001 \times 10^{-1}$$

$$= 100 \times 10^{-3} \quad \text{normalized}$$

- ▶ none of the computed digits are correct!

Catastrophic cancellation

- ▶ the error between the true value and the value computed using floating-point arithmetic is

$$\begin{aligned}|292 \times 10^{-4} - 1000 \times 10^{-4}| &= 708 \times 10^{-4} \\&= 708 \text{ ulps}\end{aligned}$$

which is very large

Catastrophic cancellation

- ▶ sometimes catastrophic cancellation can be avoided by mathematically transforming the equation to be computed into a different form so that the subtraction of similar quantities is avoided
- ▶ see *Floating-point errors* notebook for some examples

Summing many values

- ▶ because every basic floating-point arithmetic operation potentially has $\frac{1}{2}$ ulp of error, repeatedly performing an operation many times means that the final result can have a lot of accumulated error
- ▶ an example of repeating a calculation many times is computing statistics (average, standard deviation, etc) with large sample sizes

The following code fragment will print **1000125.0** instead of the true value **1000100.0**:

```
float[] arr = new float[1001];
Arrays.fill(arr, 0.1f);
arr[0] = 1_000_000f;
float sum = 0f;
for (float val : arr) {
    sum += val;
}
System.out.println(sum);
```

Simply moving the largest value to the end of the array causes the code fragment to print the true value **1000100.0**:

```
float[] arr = new float[1001];
Arrays.fill(arr, 0.1f);
arr[arr.length - 1] = 1_000_000f;
float sum = 0f;
for (float val : arr) {
    sum += val;
}
System.out.println(sum);
```

Summing many values

- ▶ sorting the values to be summed is sometimes recommended as a method of reducing rounding error when summing many values
 - ▶ changes the order of the values in the array or collection which may not be desirable
 - ▶ sorting has complexity $O(n \log n)$ whereas summing n values should have complexity $O(n)$
- ▶ a better way is to use Kahan's algorithm
 - ▶ see *Floating-point errors* notebook for details

Creating and using objects

Objects

- ▶ while it is possible to create a Java program using mostly primitive values, almost all non-trivial Java programs make extensive use of objects that interact with each other
- ▶ why use objects?
 - ▶ because objects can perform more operations than the primitive values

Object structure

- ▶ an object contains information stored in variables
 - ▶ these variables are called *fields* or *instance variables*
- ▶ the fields can be primitive types and they can be reference types
 - ▶ in other words, an object can be made up of other objects, which can be made up of other objects, which can be made up of other objects, and so on
- ▶ the set of values of an object's fields is called the *state* of the object

Object behavior

- ▶ an object can perform actions
 - ▶ the methods belonging to an object define what actions the object can perform
- ▶ objects can ask other objects to perform actions by calling methods belonging to the other objects

Object life cycle

- ▶ an object
 - 1. created
 - 2. used
 - 3. destroyed
- ▶ a Java programmer has control over 1. and 2., but not 3.

Object creation

- ▶ objects are created using the **new** operator
- ▶ the **new** operator does three things:
 1. allocates memory for the new object
 2. calls a constructor which initializes the state of the new object
 3. returns a reference to the new object

Some examples of using `new` to create `Point2` objects
(from the course notebooks)

```
// import ca.queensu.cs.cisc124.notes.basics.geometry;

Point2 p1 = new Point2();                  // (0.0, 0.0)
Point2 p2 = new Point2(1.0, 0.5);        // (1.0, 0.5)
Point2 p3 = new Point2(p2);              // (1.0, 0.5)
```

Some examples of using **new** to create **Scanner** objects.

```
// import java.util.Scanner;  
  
// read keyboard input  
Scanner s1 = new Scanner(System.in);  
  
// parse parts of a string  
String str = "1 fish 2 fish red fish blue fish";  
Scanner s2 = new Scanner(str);
```

Calling constructors

- ▶ the **new** operator is always followed by a constructor call
- ▶ the call looks like a method call, but constructors are not considered to be methods
 - ▶ in particular, constructors never return a value, not even **void**
- ▶ constructors are defined inside classes that define reference types
 - ▶ the name of a constructor *must* match the name of the class
- ▶ the purpose of a constructor is to initialize the state of a newly created object

A no-argument constructor is a constructor that has no caller specified inputs

- ▶ its purpose is to initialize a newly created object to some well-defined default state
- ▶ not every class defines a no-argument constructor

```
// import ca.queensu.cs.cisc124.notes.basics.geometry;  
  
Point2 p1 = new Point2();           // (0.0, 0.0)  
Point2 p2 = new Point2(1.0, 0.5);   // (1.0, 0.5)  
Point2 p3 = new Point2(p2);        // (1.0, 0.5)
```

A copy constructor is a constructor that has one input of the same type as the class it is defined in

- ▶ its purpose is to initialize a newly created object by copying the state of another object
- ▶ not every class defines a no-argument constructor

```
// import ca.queensu.cs.cisc124.notes.basics.geometry;  
  
Point2 p1 = new Point2();           // (0.0, 0.0)  
Point2 p2 = new Point2(1.0, 0.5);   // (1.0, 0.5)  
Point2 p3 = new Point2(p2);        // (1.0, 0.5)
```

A class may define multiple constructors

- ▶ e.g., the **Point2** class defines a constructor that lets the caller set the coordinates of the newly created point

```
// import ca.queensu.cs.cisc124.notes.basics.geometry;  
  
Point2 p1 = new Point2();           // (0.0, 0.0)  
Point2 p2 = new Point2(1.0, 0.5);   // (1.0, 0.5)  
Point2 p3 = new Point2(p2);        // (1.0, 0.5)
```

A class may define multiple constructors

- ▶ e.g., the **Scanner** class defines a constructor that lets the caller scan the keyboard for input

```
// import java.util.Scanner;  
  
// read keyboard input  
Scanner s1 = new Scanner(System.in);  
  
// parse parts of a string  
String str = "1 fish 2 fish red fish blue fish";  
Scanner s2 = new Scanner(str);
```

A class may define multiple constructors

- ▶ e.g., the **Scanner** class defines a constructor that lets the caller scan the contents of a string

```
// import java.util.Scanner;
```

```
// read keyboard input
```

```
Scanner s1 = new Scanner(System.in);
```

```
// parse parts of a string
```

```
String str = "1 fish 2 fish red fish blue fish";
```

```
Scanner s2 = new Scanner(str);
```

Using objects

- ▶ once an object has been created, it can be used via a reference to the object
- ▶ there is no way for the programmer to use an object directly in Java
- ▶ using an object usually means calling a method belonging to the object using the dot operator

referenceVariable.methodName(arguments)

Create a point, get its coordinates, and change its coordinates:

```
Point2 p = new Point2(0.5, 0.1);

double x = p.x();           // 0.5
double y = p.y();           // 0.1

x = x + 0.1;
y = y - 0.1;

p.x(x);                   // change x-coordinate to x
p.y(y);                   // change y-coordinate to y
p.set(x, y);              // change both coordinates
```

Create a point and move it back and forth.

```
public static void main(String[] args) {
    Point2 p = new Point2(0.5, 0.5);
    final double RADIUS = 0.1;
    double dx = 0.01;
    while (true) {
        double x = p.x() + dx;                      // new x coordinate
        if (x > 1.0 - RADIUS) {                      // right edge of window
            x = 1.0 - RADIUS;
            dx = -dx;
        }
        else if (x < 0.0 + RADIUS) {                // left edge of window
            x = 0.0 + RADIUS;
            dx = -dx;
        }
        p.x(x);                                     // move point
        StdDraw.clear();                            // draw point
        StdDraw.filledCircle(p.x(), p.y(), RADIUS);
        StdDraw.show(15);
    }
}
```

See `ca.queensu.cs.cisc124.notes.basics.geometry.BouncingBall`
for a more sophisticated example.

Create a **Scanner** object and echo back what the user types:

```
import java.util.Scanner;

public class ScannerExample {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            System.out.println("You entered: " + line);
        }
        sc.close();
    }
}
```

Create a **Scanner** object and echo back what the user types:

```
import java.util.Scanner;

public class ScannerExample {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        while (sc.hasNextLine()) {
            String line = sc.nextLine();
            if (line.equals("quit")) {
                break;
            }
            System.out.println("You entered: " + line);
        }
        sc.close();
    }
}
```

Reading APIs

- ▶ classes are documented by their API (application programming interface)
- ▶ the Java Standard Library classes are documented online:
 - ▶ <https://docs.oracle.com/javase/8/docs/api/index.html>
 - ▶ alternatively use a search engine to search for the documentation for a class
- ▶ the notebook classes are documented in their eclipse project under the **doc** folder
- ▶ reading API example here

Java Lists

Lists

- ▶ Java lists provide similar functionality to lists in Python, but the syntax is very different
- ▶ also, Java allows the programmer to choose between several different kinds of lists
 - ▶ but there are two main kinds, and you usually use one kind in particular
 - ▶ **ArrayList**

Element type and lists

- ▶ one significant difference of Java lists compared to Python lists is that a Java list almost always holds elements of one type
- ▶ the type is specified inside `<>` when declaring a list variable and calling a list constructor
 - ▶ only use this notation when working with what are called *generic classes*

Create a list of strings:

```
ArrayList<String> t = new ArrayList<String>();
```

```
// after Java 7 you can usually omit the type on  
// the constructor call
```

```
ArrayList<String> t = new ArrayList<>();
```

No lists of primitive elements

- ▶ one limitation of Java's generic type mechanism is that it is impossible to use a primitive type as a generic type
 - ▶ e.g., no `ArrayList<int> d`

Wrapper classes

- ▶ every primitive type has a corresponding reference type defined by a wrapper class

Primitive type	Wrapper class
<code>boolean</code>	<code>Boolean</code>
<code>byte</code>	<code>Byte</code>
<code>char</code>	<code>Character</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>

Wrapper classes

- ▶ similar to the String class, the wrapper classes receive special treatment by the compiler
- ▶ in particular, the compiler will try to convert between a value from its primitive type and its wrapper class and vice versa when required

```
int i = 1;  
Integer j = i;      // ok  
int k = j;         // ok
```

When you want a list of some primitive type element,
create a list using the wrapper type:

```
ArrayList<Integer> t = new ArrayList<>();  
ArrayList<Double> u = new ArrayList<>();  
ArrayList<Character> v = new ArrayList<>();
```

Python to Java

Python	Java
t = []	ArrayList<String> t = new ArrayList<>();
if t:	if (t.isEmpty()) {
sz = len(t)	int sz = t.size();
t.append('hello')	t.add("hello");
e = t[0]	String e = t.get(0);
t[0] = 'goodbye'	t.set(0, "goodbye");
if val in t:	if (t.contains(val)) {
t.extend(another_list)	t.addAll(anotherList);
e = t.pop(index)	String e = t.remove(index);
e = t.remove(elem)	String e = t.remove(elem);
u = t[start:stop]	List<String> u = t.subList(start, stop);

Java Lists

Lists

- ▶ Java lists provide the functionality of the list abstract data type
- ▶ Java allows the programmer to choose between several different kinds of list implementations
 - ▶ but there are two main kinds, and you usually use one kind in particular
 - ▶ **java.util.ArrayList**
 - ▶ <https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

Element type and lists

- ▶ one significant difference of Java lists compared to Python lists is that a Java list almost always holds elements of one type
- ▶ the type is specified inside `<>` when declaring a list variable and calling a list constructor
 - ▶ only use this notation when working with what are called *generic classes*

Create a list of strings:

```
ArrayList<String> t = new ArrayList<String>();
```

```
// after Java 7 you can usually omit the type on  
// the constructor call  
ArrayList<String> t = new ArrayList<>();
```

No lists of primitive elements

- ▶ one limitation of Java's generic type mechanism is that it is impossible to use a primitive type as a generic type
 - ▶ e.g., no `ArrayList<int> d`

Wrapper classes

- ▶ every primitive type has a corresponding reference type defined by a wrapper class

Primitive type	Wrapper class
<code>boolean</code>	<code>Boolean</code>
<code>byte</code>	<code>Byte</code>
<code>char</code>	<code>Character</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>

Wrapper classes

- ▶ similar to the String class, the wrapper classes receive special treatment by the compiler
- ▶ in particular, the compiler will try to convert between a value from its primitive type and its wrapper class and vice versa when required

```
int i = 1;  
Integer j = i;      // ok, called auto-boxing  
int k = j;         // ok, called auto-unboxing
```

When you want a list of some primitive type element,
create a list using the wrapper type:

```
ArrayList<Integer> t = new ArrayList<>();  
ArrayList<Double> u = new ArrayList<>();  
ArrayList<Character> v = new ArrayList<>();
```

When you use a list where the elements are a wrapper type, you can almost always pretend that the list contains primitive type elements

- ▶ the **add** method appends an element to the end of the list
- ▶ the **get** method returns the element at the specified index

```
ArrayList<Integer> t = new ArrayList<>();  
t.add(100);           // add element 100 to the list  
                     // compiler autoboxes 100 to Integer  
  
int first = t.get(0);    // get 100 from the list  
                     // compiler auto-unboxes element  
                     // to int
```

The one situation where this does not work is when using the **remove** method with a list of **Integers**

- ▶ does not work as intended because there are two versions of **remove** and one version has an **int** as its parameter

```
ArrayList<Integer> t = new ArrayList<>();  
// add some elements here  
  
// try to remove the element with value 100?  
t.remove(100);
```

This actually removes the element at index **100**

- ▶ to remove the first element equal to **100** from the list, you must use an **Integer** object

```
ArrayList<Integer> t = new ArrayList<>();  
// add some elements here  
  
// removes the first element with value 100  
t.remove(Integer.valueOf(100));
```

List operations

	Example
no-argument constructor	ArrayList<String> t = new ArrayList<>();
is the list empty?	if (t.isEmpty()) {
number of elements in the list	int sz = t.size();
append to the end of the list	t.add("hello");
get an element by index	String e = t.get(0);
set an element by index	t.set(0, "goodbye");
does the list contain a specified value?	if (t.contains(val)) {
append the contents of another list to the end of a list	t.addAll(anotherList);
remove by index	String e = t.remove(index);
remove the first element equal to a specified element	String e = t.remove(elem);

List operations

Python	Java
get a sublist by start and stop indexes	<code>List<String> u = t.subList(start, stop);</code>
make a new copy of a list	<code>ArrayList<String> u = new ArrayList<>(t);</code>

List operations

Python	Java
count the number of times an element appears in a list	<code>int n = Collections.frequency(t, "abc");</code>
reverse the order of the elements in a list	<code>Collections.reverse(t);</code>
sort a list	<code>Collections.sort(t); // or</code> <code>t.sort(null);</code>

- ▶ the class **java.util.Collections** contains many methods that operate on lists
- ▶ <https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>

Iterating over the elements of a list is almost identical to iterating over the elements of an array:

```
// assume that t is a non-empty ArrayList<String>
// find the shortest string in t

String shortest = t.get(0);
int minLength = shortest.length();
for (String s : t) {
    if (s.length() < minLength) {
        minLength = s.length();
        shortest = s;
    }
}
```

Iterating over the elements of a list is almost identical to iterating over the elements of an array:

```
// assume that t is a non-empty ArrayList<String>
// find the shortest string in t

String shortest = t.get(0);
int minLength = shortest.length();
for (int i = 1; i < t.size(); i++) {
    String s = t.get(i);
    if (s.length() < minLength) {
        minLength = s.length();
        shortest = s;
    }
}
```

We can create a method that finds the shortest string in a list that tests for an input error:

```
public static String shortest(ArrayList<String> t) {  
    if (t.isEmpty()) {  
        throw new IllegalArgumentException("empty list");  
    }  
    String shortest = t.get(0);          // ok, list not empty  
    int minLength = shortest.length();  
    for (String s : t) {  
        if (s.length() < minLength) {  
            minLength = s.length();  
            shortest = s;  
        }  
    }  
    return shortest;  
}
```

Simulating odds

- Given two six-sided, fair dice, what are the odds of rolling a 2? A 3? A 4?

Sum of dice values	Probability
2	2.78%
3	5.56%
4	8.33%
5	11.11%
6	13.89%
7	16.67%
8	13.89%
9	11.11%
10	8.33%
11	5.56%
12	2.78%

Rolling dice

- ▶ the class **java.util.Random** allows the programmer to create objects that can generate random **boolean**, **int**, **long**, **float**, and **double** values
- ▶ https://en.wikipedia.org/wiki/Linear_congruential_generator
- ▶ <https://docs.oracle.com/javase/8/docs/api/java/util/Random.html>

Simulating odds

- ▶ to compute the odds via simulation, perform the following steps
 1. Create a list and add 13 zeros to the list
 2. Repeat the following n times:
 1. Simulate rolling two dice and sum their values to get x
 2. Add 1 to the value at index x in the list
 3. Divide each element of the list by n
- ▶ solve problem using eclipse here

Coin-flipping simulation

- ▶ If you flip a fair coin n times how many heads do you expect to see? How often do you expect to see $\frac{n}{4}$ heads?
- ▶ https://en.wikipedia.org/wiki/Binomial_distribution

Number of heads	Probability
0	
1	
2	
...	
...	
n	

Coin flipping simulation

- ▶ to compute the odds via simulation, perform the following steps

1. Create a list and add $n + 1$ zeros to the list
2. Repeat the following m times:
 1. $count = 0$
 2. Repeat the following n times:
 1. Simulate flipping a coin
 2. if heads
 - $count = count + 1$
 3. Add 1 to the value at index $count$ in the list
 3. Divide each element of the list by m

- ▶ solve problem using eclipse here

Coin-flipping simulation

- ▶ If you flip a fair coin n times how many heads do you expect to see? How often do you expect to see $\frac{n}{4}$ heads?
- ▶ https://en.wikipedia.org/wiki/Binomial_distribution

Number of heads	Probability
0	
1	
2	
...	
...	
n	

Coin flipping simulation

- ▶ to compute the odds via simulation, perform the following steps

1. Create a list and add $n + 1$ zeros to the list
2. Repeat the following m times:
 1. $count = 0$
 2. Repeat the following n times:
 1. Simulate flipping a coin
 2. if heads
 - $count = count + 1$
 3. Add 1 to the value at index $count$ in the list
 3. Divide each element of the list by m

- ▶ solve problem using eclipse here

```
import java.util.ArrayList;
import java.util.Random;

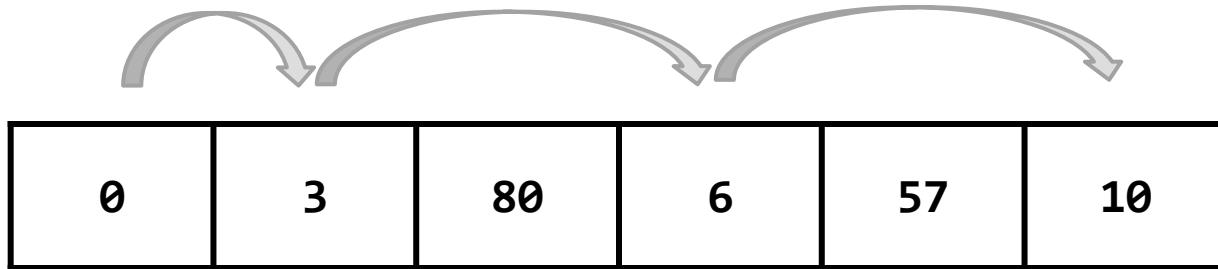
public class CoinFlip {
    public static void main(String[] args) {
        final int N = 20;
        final int M = 1000000;
        ArrayList<Double> prob = new ArrayList<>();
        for (int i = 0; i <= N; i++) {
            prob.add(0.0);
        }
        Random rng = new Random();
        for (int i = 0; i < M; i++) {
            int count = 0;
            for (int j = 0; j < N; j++) {
                if (rng.nextBoolean()) {
                    count++;
                }
            }
            prob.set(count, prob.get(count) + 1);
        }
        double cumProb = 0.0;
        for (int i = 0; i <= N; i++) {
            prob.set(i, prob.get(i) / M);
            cumProb += prob.get(i);
        }
        System.out.println(prob);
        System.out.println(cumProb);
    }
}
```

Jump It

0	3	80	6	57	10
---	---	----	---	----	----

- ▶ board of n squares, $n \geq 2$
- ▶ start at the first square on left
- ▶ on each move you can move 1 or 2 squares to the right
- ▶ each square you land on has a cost (the value in the square)
 - ▶ costs are always positive
- ▶ goal is to reach the rightmost square with the lowest cost

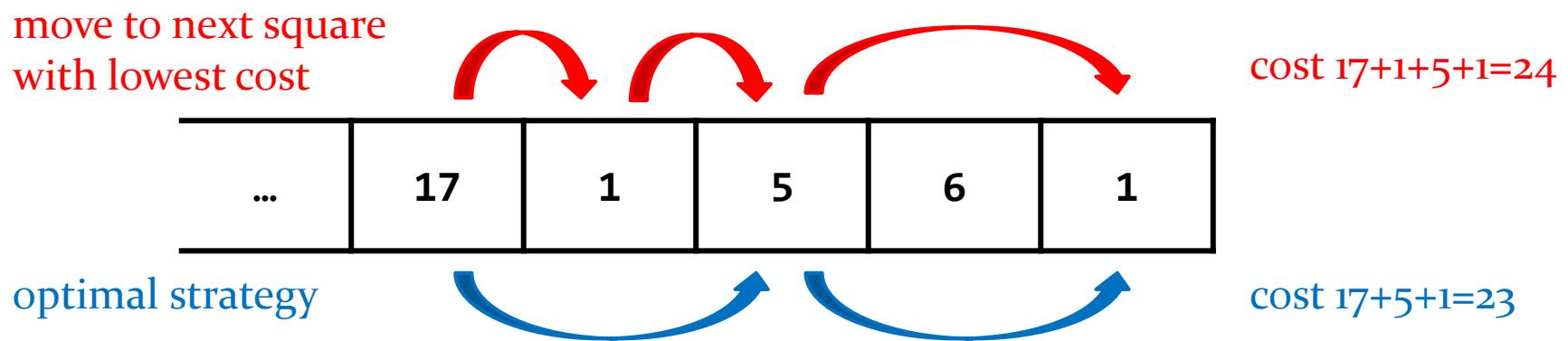
Jump It



- ▶ solution for example:
 - ▶ move 1 square
 - ▶ move 2 squares
 - ▶ move 2 squares
 - total cost = $0 + 3 + 6 + 10 = 19$
- ▶ can the problem be solved by always moving to the next square with the lowest cost?

Jump It

- no, it might be better to move to a square with higher cost because you would have ended up on that square anyway



Jump It

- ▶ sketch a small example of the problem
 - ▶ it will help you find the base cases
 - ▶ it might help you find the recursive cases

Jump It

- ▶ base case(s):
 - ▶ `board.size() == 2`
 - ▶ no choice of move (must move 1 square)
 - ▶ `cost = board.get(0) + board.get(1);`
 - ▶ `board.size() == 3`
 - ▶ move 2 squares (avoiding the cost of 1 square)
 - ▶ `cost = board.get(0) + board.get(2);`

Jump It

```
public static int cost(List<Integer> board) {  
    if (board.size() == 2) {  
        return board.get(0) + board.get(1);  
    }  
    if (board.size() == 3) {  
        return board.get(0) + board.get(2);  
    }  
}
```

Jump It

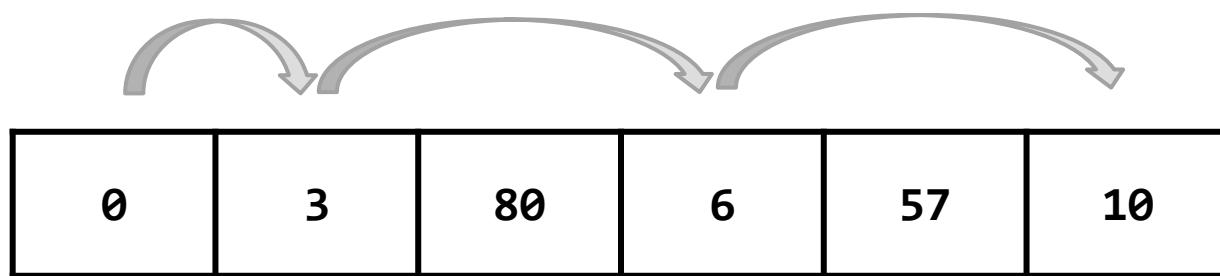
- ▶ recursive case(s):
 - ▶ compute the cost of moving 1 square
 - ▶ compute the cost of moving 2 squares
- ▶ return the smaller of the two costs

Jump It

```
public static int cost(List<Integer> board) {  
    if (board.size() == 2) {  
        return board.get(0) + board.get(1);  
    }  
    if (board.size() == 3) {  
        return board.get(0) + board.get(2);  
    }  
    List<Integer> afterOneStep = board.subList(1, board.size());  
    List<Integer> afterTwoStep = board.subList(2, board.size());  
    int c = board.get(0);  
    return c + Math.min(cost(afterOneStep), cost(afterTwoStep));  
}
```

Jump It

- ▶ can you modify the `cost` method so that it also produces a list of moves?
- ▶ e.g., for the following board



the method produces the list [1, 2, 2]

- ▶ consider using the following modified signature

```
public static int cost(List<Integer> board, List<Integer> moves)
```

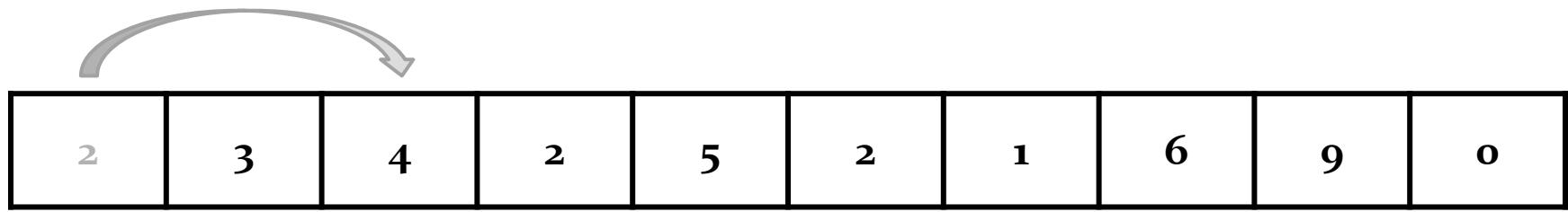
-
- ▶ the Jump It problem has a couple of nice properties:
 - ▶ the rules of the game make it impossible to move to the same square twice
 - ▶ the rules of the games make it impossible to try to move off of the board
 - ▶ consider the following problem

-
- ▶ given a list of non-negative integer values:

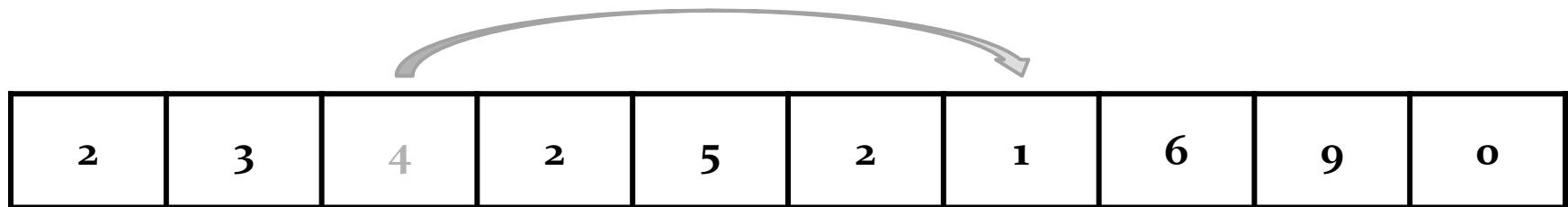
2	3	4	2	5	2	1	6	9	0
---	---	---	---	---	---	---	---	---	---

- ▶ starting from the first element try to reach the last element (whose value is always zero)
- ▶ you may move left or right by the number of elements equal to the value of the element that you are currently on
- ▶ you may not move outside the bounds of the list

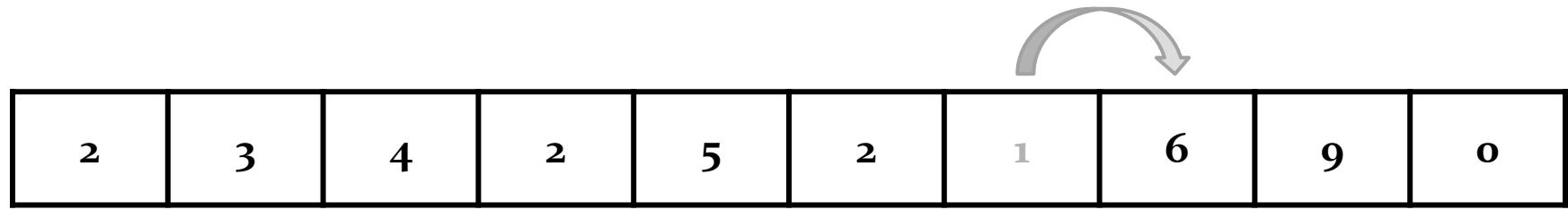
Solution 1



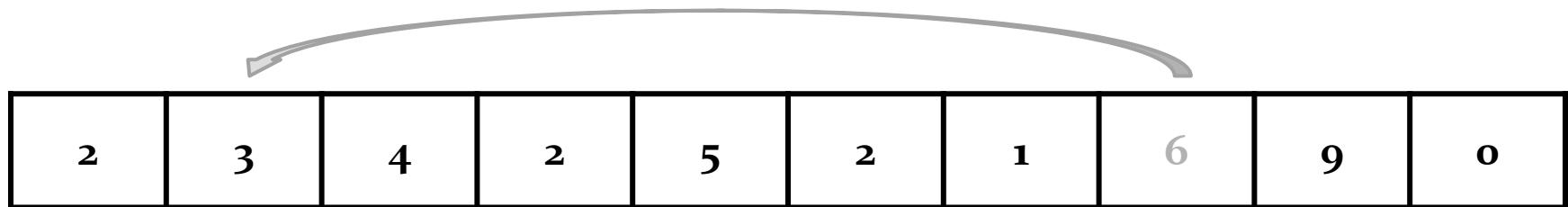
Solution 1



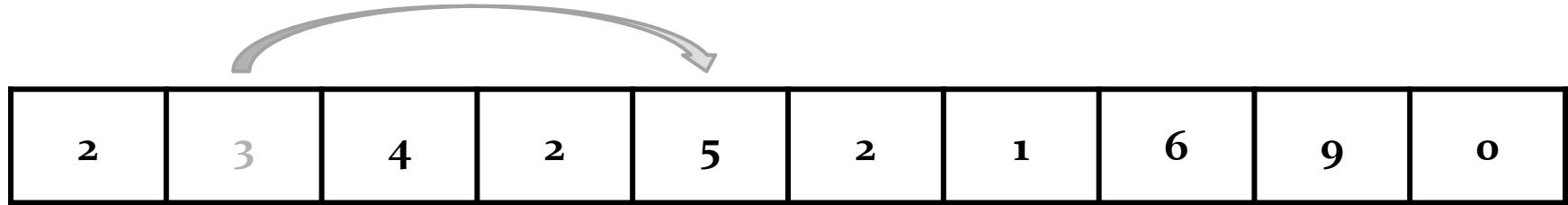
Solution 1



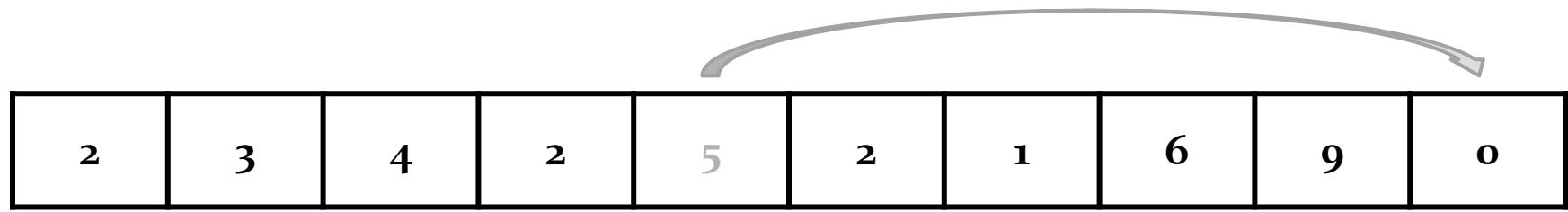
Solution 1



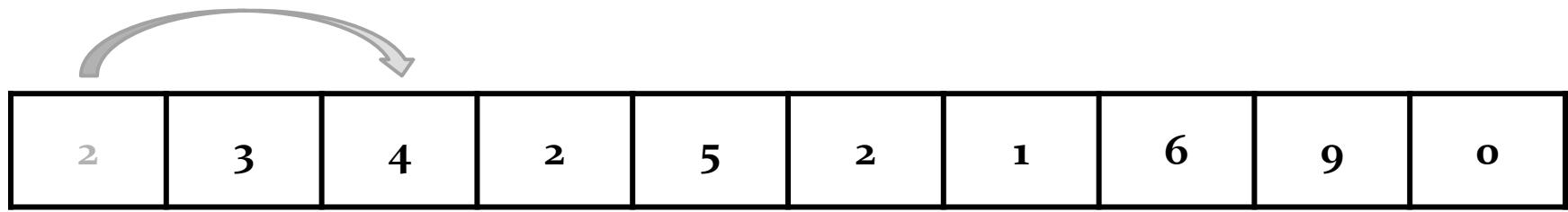
Solution 1



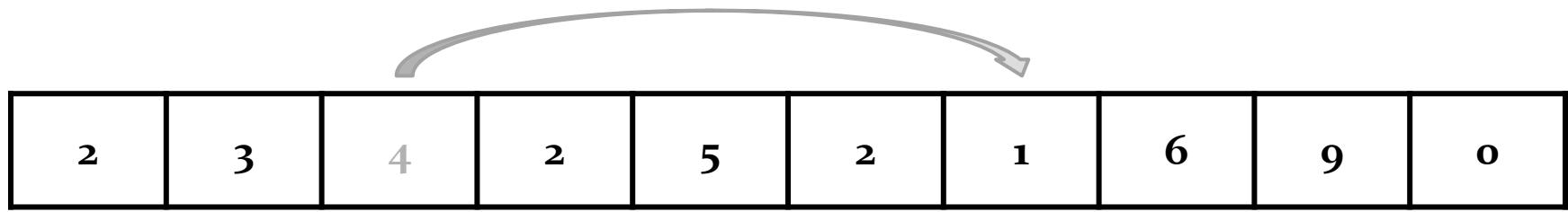
Solution 1



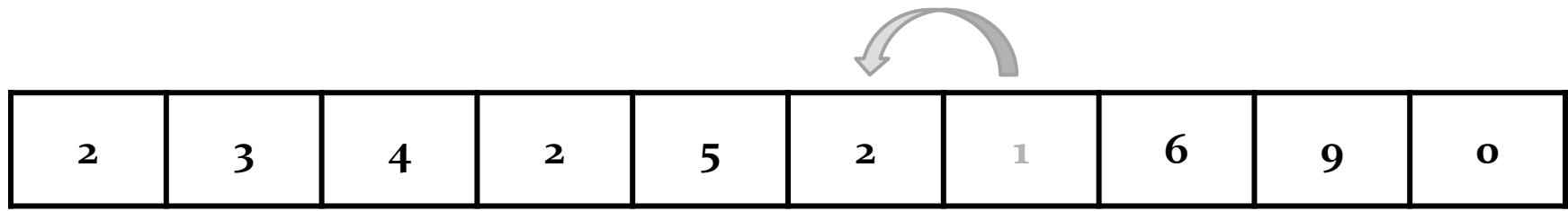
Solution 2



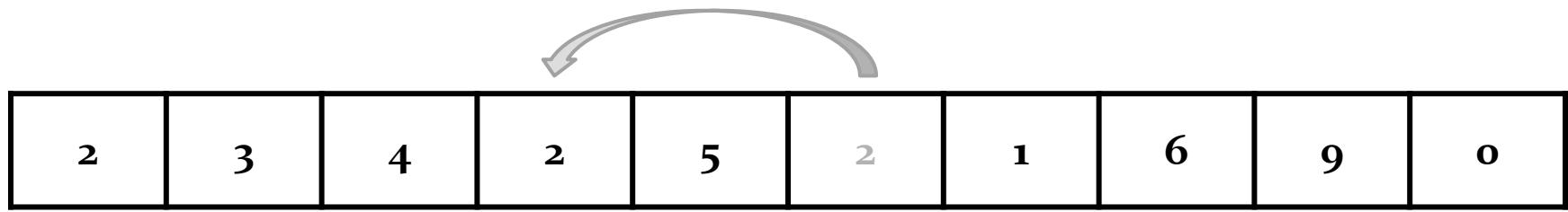
Solution 2



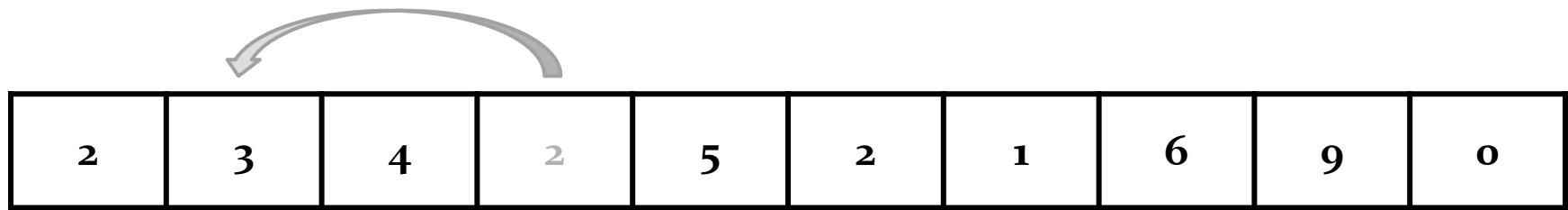
Solution 2



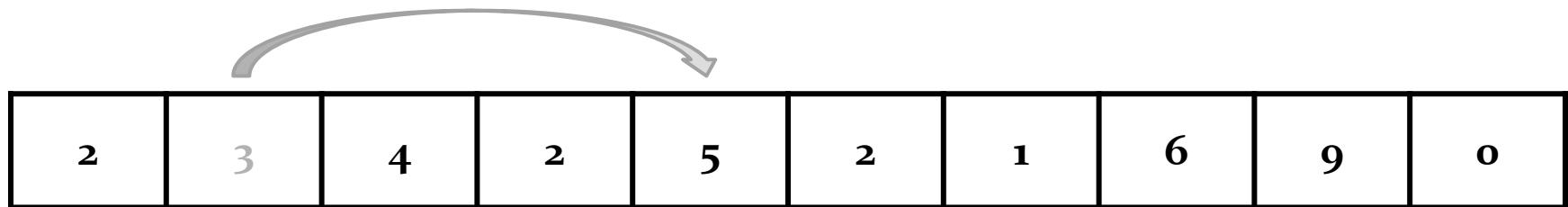
Solution 2



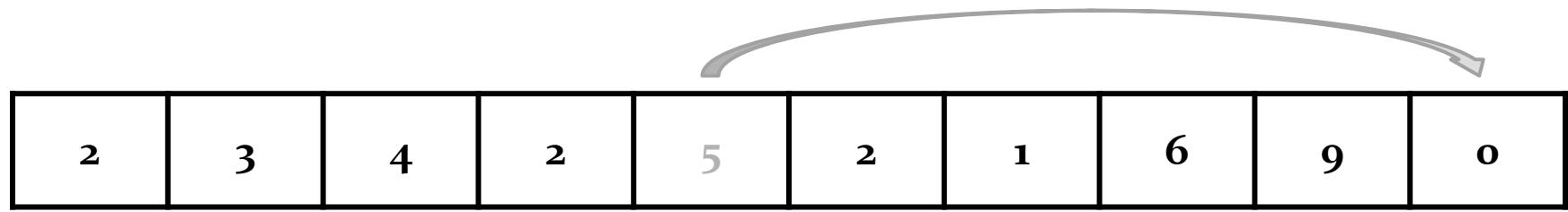
Solution 2



Solution 2

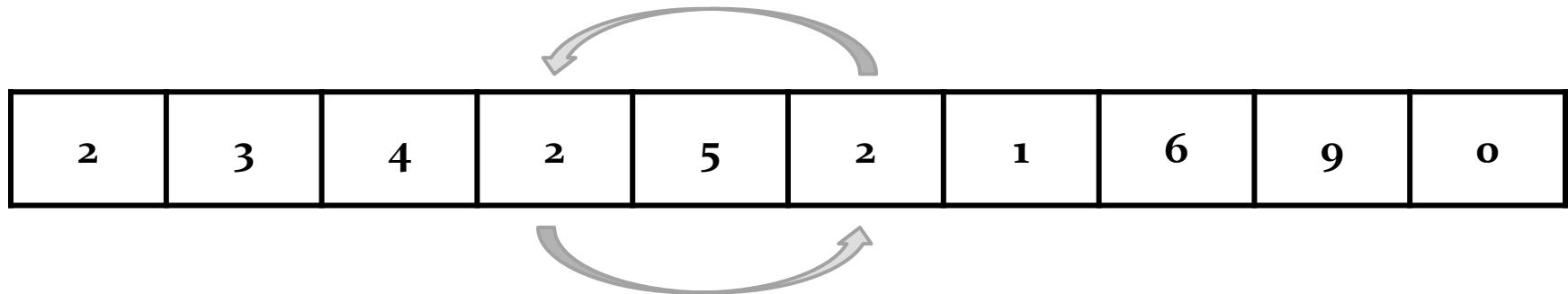


Solution 2



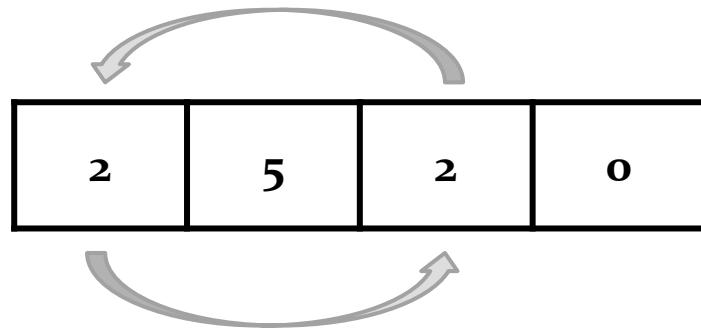
Cycles

- it is possible to find cycles where a move takes you back to a square that you have already visited



Cycles

- ▶ using a cycle, it is easy to create a board where no solution exists



Cycles

- ▶ on the board below, no matter what you do, you eventually end up on the 1 which leads to a cycle

2	1	2	2	2	2	2	6	3	0
---	---	---	---	---	---	---	---	---	---

No Solution

- ▶ even without using a cycle, it is easy to create a board where no solution exists

1	100	2	0
---	-----	---	---

-
- ▶ unlike Jump It, the board does not get smaller in an obvious way after each move
 - ▶ but it does in fact get smaller (otherwise, a recursive solution would never terminate)
 - ▶ how does the board get smaller?
 - ▶ how do we indicate this?

Recursion

- ▶ recursive cases:
 - ▶ can we move left without falling off of the board?
 - ▶ if so, can the board be solved by moving to the left?
 - ▶ can we move right without falling off of the board?
 - ▶ if so, can the board be solved by moving to the right?

```
/**  
 * Is a board is solvable when the current move is at location  
 * index of the board? The method does not modify the board.  
 *  
 * @param index  
 *         the current location on the board  
 * @param board  
 *         the board  
 * @return true if the board is solvable, false otherwise  
 */  
  
public static boolean isSolvable(int index, List<Integer> board) {  
}
```

```
public static boolean isSolvable(int index, List<Integer> board) {  
    // base cases here  
    int value = board.get(index);  
    // remember to unset board before returning  
    board.set(index, -1);  
  
}  
}
```

```
public static boolean isSolvable(int index, List<Integer> board) {  
    // base cases here  
    int value = board.get(index);  
    // remember to unset board before returning  
    board.set(index, -1);  
    boolean winLeft = false;  
    if ((index - value) >= 0) {  
  
    }  
}
```

```
public static boolean isSolvable(int index, List<Integer> board) {  
    // base cases here  
    int value = board.get(index);  
    // remember to unset board before returning  
    board.set(index, -1);  
  
    boolean winLeft = false;  
    if ((index - value) >= 0) {  
        winLeft = isSolvable(index - value, board);  
    }  
}
```

```
public static boolean isSolvable(int index, List<Integer> board) {  
    // base cases here  
    int value = board.get(index);  
    // remember to unset board before returning  
    board.set(index, -1);  
    boolean winLeft = false;  
    if ((index - value) >= 0) {  
        winLeft = isSolvable(index - value, copy);  
    }  
  
    boolean winRight = false;  
    if ((index + value) < board.size()) {  
    }  
}
```

```
public static boolean isSolvable(int index, List<Integer> board) {  
    // base cases here  
    int value = board.get(index);  
    // remember to unset board before returning  
    board.set(index, -1);  
    boolean winLeft = false;  
    if ((index - value) >= 0) {  
        winLeft = isSolvable(index - value, copy);  
    }  
}
```

```
boolean winRight = false;  
if ((index + value) < board.size()) {  
    winRight = isSolvable(index + value, board);  
}
```

```
}
```

```
public static boolean isSolvable(int index, List<Integer> board) {  
    // base cases here  
    int value = board.get(index);  
    // remember to unset board before returning  
    board.set(index, -1);  
    boolean winLeft = false;  
    if ((index - value) >= 0) {  
        winLeft = isSolvable(index - value, copy);  
    }  
  
    boolean winRight = false;  
    if ((index + value) < board.size()) {  
        winRight = isSolvable(index + value, board);  
    }  
    // unset value at index  
    board.set(index, value);  
    return winLeft || winRight;  
}
```

Base Cases

- ▶ base cases:
 - ▶ we've reached a square whose value is -1
 - ▶ board is not solvable
 - ▶ we've reached the last square
 - ▶ board is solvable

```
public static boolean isSolvable(int index, List<Integer> board) {  
    if (board.get(index) < 0) {  
        return false;  
    }  
    if (index == board.size() - 1) {  
        return true;  
    }  
    // recursive cases go here...  
}
```

Recursively Move Smallest to Front

- ▶ how do you write a method that moves the smallest element in a list to the front of the list using recursion?

Recursively Move Smallest to Front

8	7	6	4	3	5	0	2	9	1
---	---	---	---	---	---	---	---	---	---

original list

8	7	6	4	3	5	0	2	9	1
---	---	---	---	---	---	---	---	---	---

recursion

move the smallest element of this sublist
to the front of the sublist

8	0
---	---	-----	-----	-----	-----	-----	-----	-----	-----

compare



compare these two elements and move the
smallest one to the front (swapping positions)

0	8
---	---	-----	-----	-----	-----	-----	-----	-----	-----

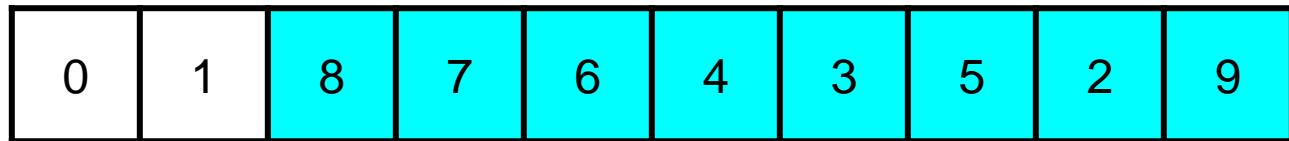
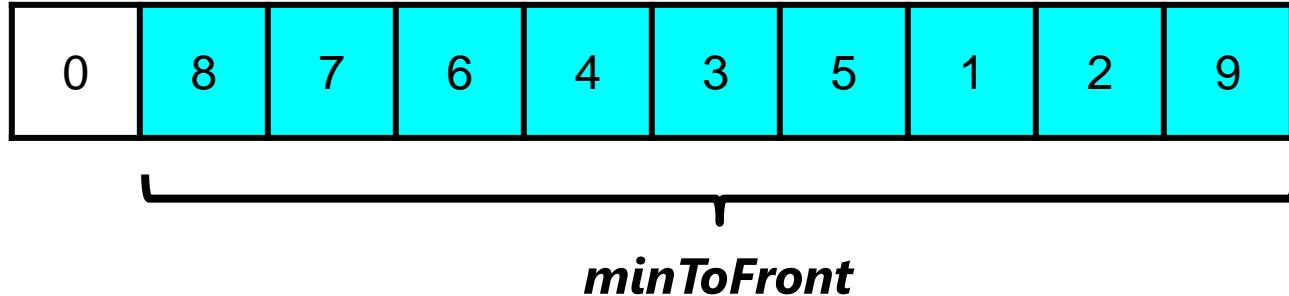
updated list

Recursively Move Smallest to Front

```
public class Sort {  
  
    public static void minToFront(List<Integer> t) {  
        if (t.size() < 2) {  
            return;  
        }  
        Sort.minToFront(t.subList(1, t.size()));  
        int first = t.get(0);  
        int second = t.get(1);  
        if (second < first) {  
            t.set(0, second);  
            t.set(1, first);  
        }  
    }  
}
```

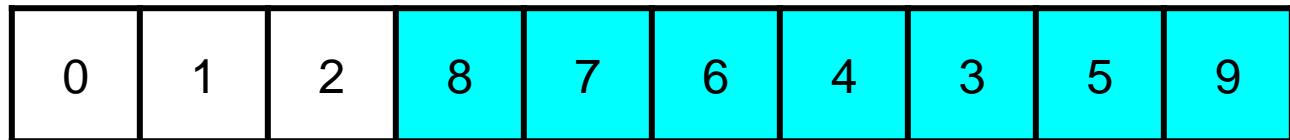
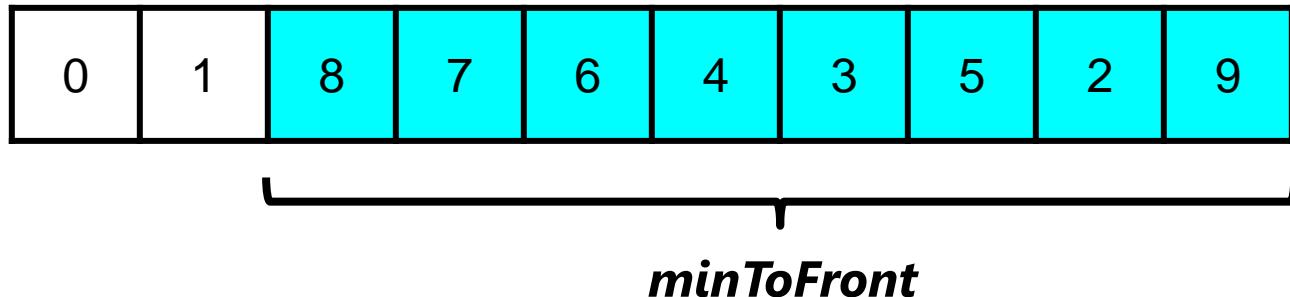
Sorting the List

- ▶ observe what happens if you repeat the process with the sublist made up of the second through last elements:



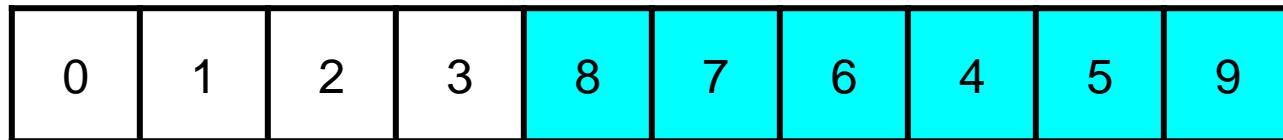
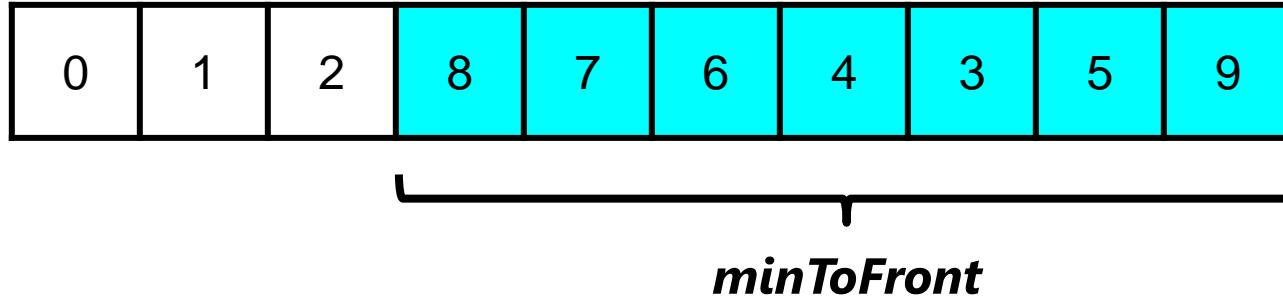
Sorting the List

- ▶ observe what happens if you repeat the process with the sublist made up of the third through last elements:



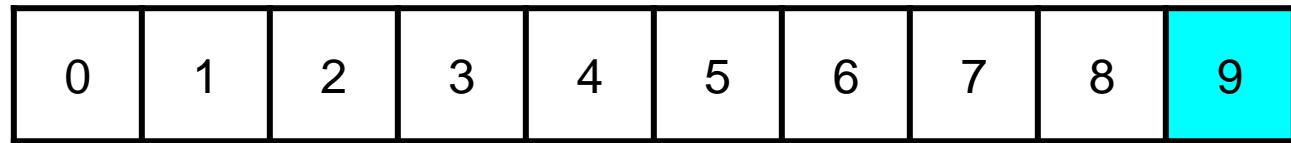
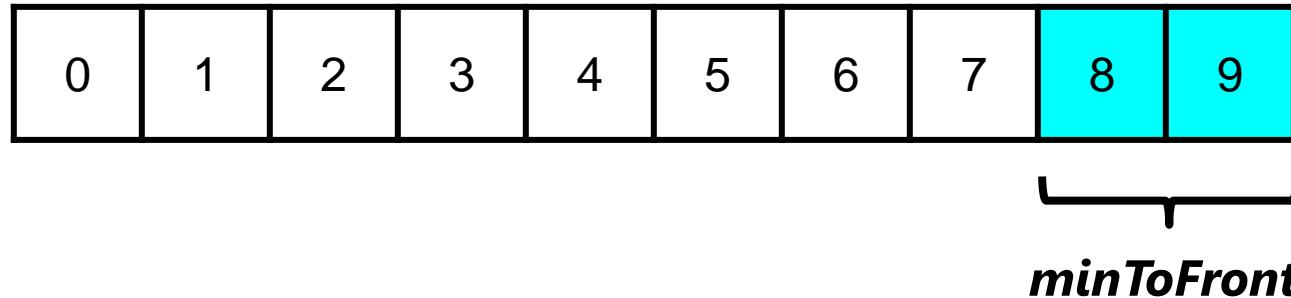
Sorting the List

- ▶ observe what happens if you repeat the process with the sublist made up of the fourth through last elements:



Sorting the List

- if you keep calling `minToFront` until you reach a sublist of size two, you will sort the original list:



- this is the *selection sort* algorithm

Selection Sort

```
public class Sort {
```

```
// minToFront not shown
```

```
public static void selectionSort(List<Integer> t) {
```

```
    if (t.size() > 1) {
```

```
        Sort.minToFront(t);
```

```
        Sort.selectionSort(t.subList(1, t.size()));
```

```
}
```

```
}
```

```
}
```

Selection Sort

- ▶ there are only two steps in the selection sort algorithm
 1. move the smallest element in the list to the front
 - ▶ this has complexity $O(n)$
 2. recursively selection sort the sublist of size $(n - 1)$
- ▶ let $T(n)$ be the number of operations needed to selection sort a list of size n
- ▶ then the recurrence relation is:

$$T(n) = T(n - 1) + O(n)$$

- ▶ solving the recurrence results in

$$T(n) = O(n^2)$$

Sets

Sets

- ▶ a set is a collection of unique elements
 - ▶ no two elements are equal to one another
- ▶ common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, and difference
 - ▶ in Java, knowledge of sets is needed to effectively use maps
- ▶ no indexing or slicing using an index
 - ▶ but some types of Java sets support returning a subset of a set

Sets

- ▶ Java sets provide similar functionality to the set abstract data type
- ▶ also, Java allows the programmer to choose between several different kinds of sets
 - ▶ but there are two main kinds
 - ▶ **java.util.HashSet**
 - <https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html>
 - most similar to a Python set
 - ▶ **java.util.TreeSet**
 - <https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html>
 - a set that sorts its elements

Element type and sets

- ▶ Java sets are generic and can hold elements of a same type
- ▶ the type is specified inside <> when declaring a set variable and calling a set constructor
 - ▶ only use this notation when working with what are called *generic classes*

Create a set of strings:

```
HashSet<String> t = new HashSet<String>();
```

```
// after Java 7 you can usually omit the type on  
// the constructor call
```

```
HashSet<String> t = new HashSet<>();
```

No sets of primitive elements

- ▶ one limitation of Java's generic type mechanism is that it is impossible to a primitive type as a generic type
 - ▶ must use a wrapper class type (same as lists)
 - ▶ e.g., no **HashSet<int>**, use **HashSet<Integer>** instead

Java methods common to all sets

	Java
create an empty set	HashSet<String> t = new HashSet<>();
is the set empty?	if (t.isEmpty()) {
number of elements in the set	int sz = t.size();
add an element to the set (location in the set is unknown to the caller)	t.add("hello");
does the set contain an element equal to a specified value?	if (t.contains(val)) {
remove an element equal to elem	boolean boo = t.remove(elem);
copy a set	HashSet<String> u = new HashSet<>(t);

Creating a set

- ▶ to create a set, you usually choose between two different kinds (but other kinds exist)
- ▶ usually, you will want to use **HashSet**
 - ▶ very fast **add**, **remove**, **contains**
 - $O(1)$ expected complexity
 - ▶ makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time
- ▶ the other main choice is **TreeSet**
 - ▶ fast **add**, **remove**, **contains** (but slower than **HashSet**)
 - $O(\log n)$ complexity
 - ▶ elements are ordered by their natural ordering (sorted)
 - ▶ elements must have type that implements **Comparable** or the set must be initialized with a **Comparator** object

Creating a set

- ▶ a third kind that is occasionally useful is **LinkedHashSet**
 - ▶ very fast **add**, **remove**, **contains**
 - $O(1)$ expected complexity (but slightly slower than **HashSet**)
 - ▶ iteration order is guaranteed to be identical to the order in which elements were added to the set
 - ▶ costs more memory than **HashSet**
 - additional cost is proportional to the number of elements in the set

Iterating over a set

- ▶ sets do not support indexing, thus iterating over a set is done using a for-each loop

```
// assume t is some kind of set of strings  
  
for (String s : t) {  
    // do something with s here  
}
```

Uniqueness of elements

- ▶ suppose you have a list and you want to find all of the elements that occur one or more times; you can use a set to remove the duplicated elements

```
// csAcct is a list of CS student account names  
// ArrayList<String> csAcct = ...
```

```
HashSet<String> uniqueAcct = new HashSet<>(csAcct);
```

- ▶ use a **TreeSet** if you want a sorted set of accounts

```
// csAcct is a list of CS student account names  
// ArrayList<String> csAcct = ...
```

```
TreeSet<String> uniqueAcct = new TreeSet<>(csAcct);
```

Uniqueness of elements

- ▶ use a **LinkedHashSet** if you want the set to maintain the same ordering of elements as the input list

```
// csAcct is a list of CS student account names  
// ArrayList<String> csAcct = ...
```

```
LinkedHashSet<String> uniqueAcct = new LinkedHashSet<>(csAcct);
```

Uniqueness of elements

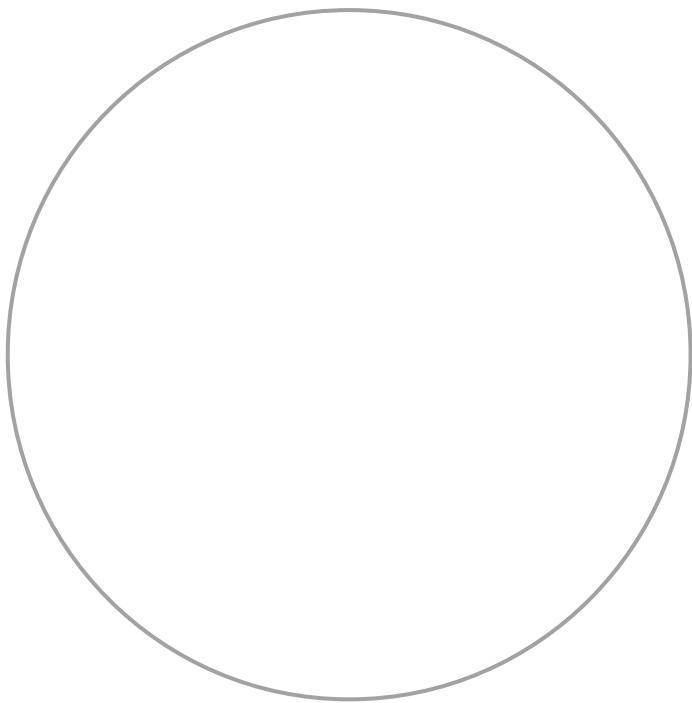
- ▶ if you want to find the duplicated elements of a collection, you can use a pair of sets

```
// csAcct is a list of CS student account names

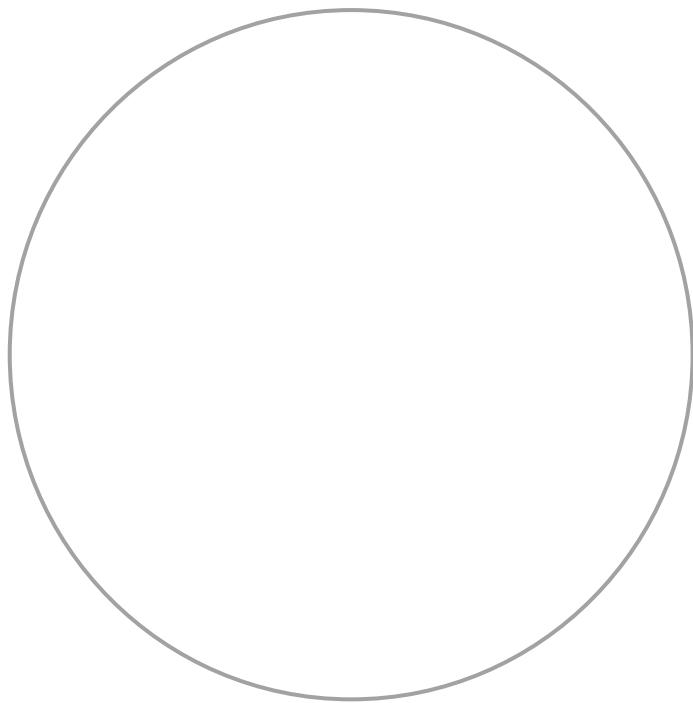
HashSet<String> uniqueAcct = new HashSet<String>();
HashSet<String> duplicateAcct = new HashSet<String>();

for (String acct : csAcct) {
    if (!uniqueAcct.add(acct)) {
        duplicateAcct.add(acct);
    }
}
```

```
csAcct: [“spkouma”, “dperit”, “hp1304”, “arian98”,  
“spkouma”, “dperit”]
```



uniqueAcct

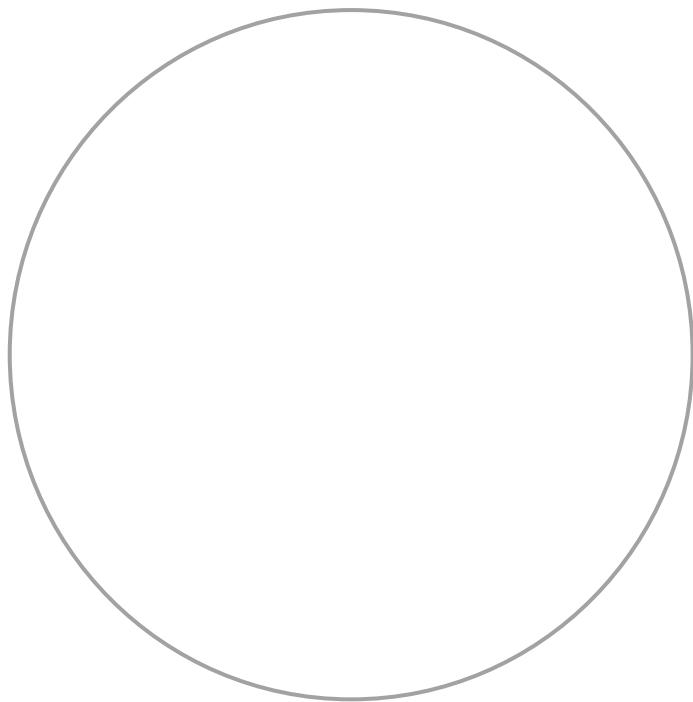


duplicatedAcct

```
csAcct: [“spkouma”, “dperit”, “hp1304”, “arian98”,  
“spkouma”, “dperit”]
```

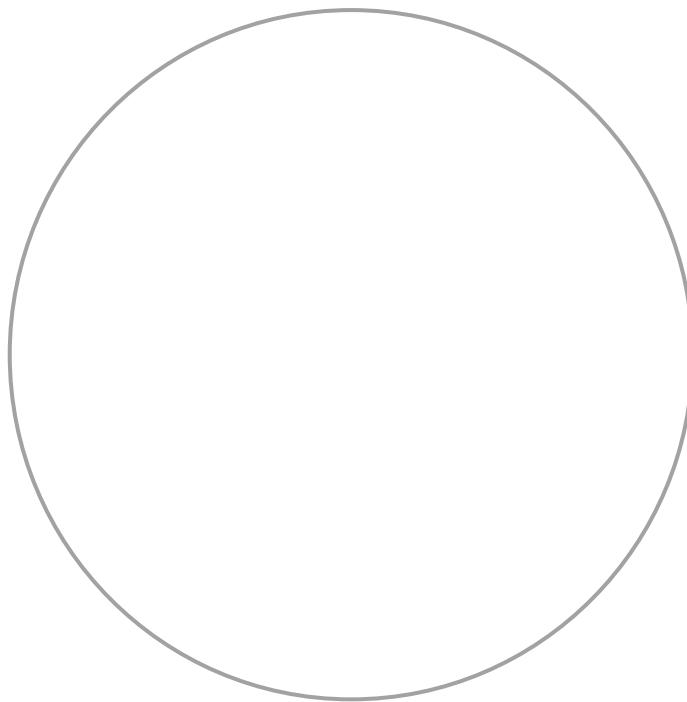


uniqueAcct

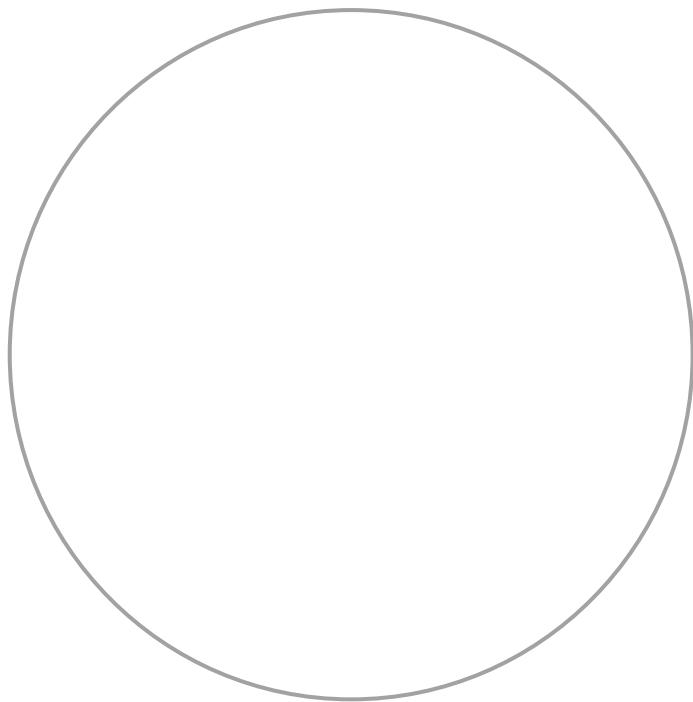
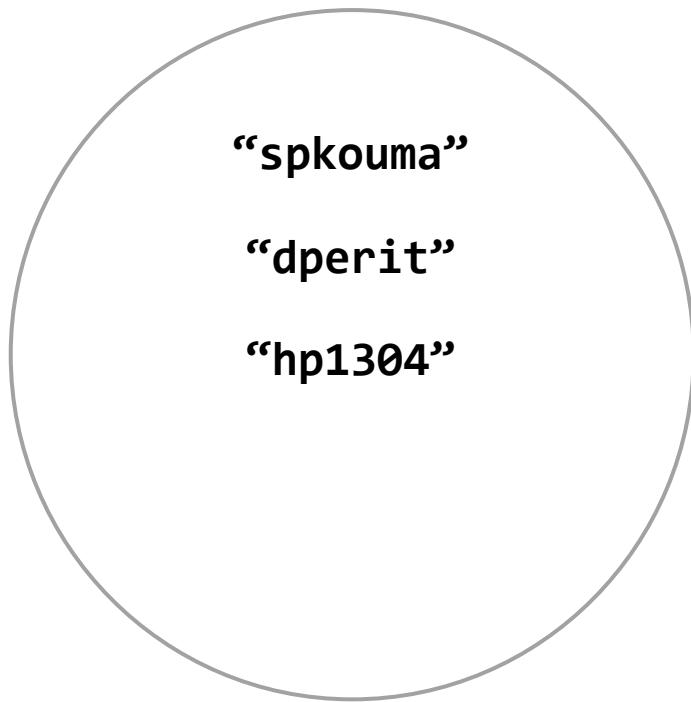


duplicatedAcct

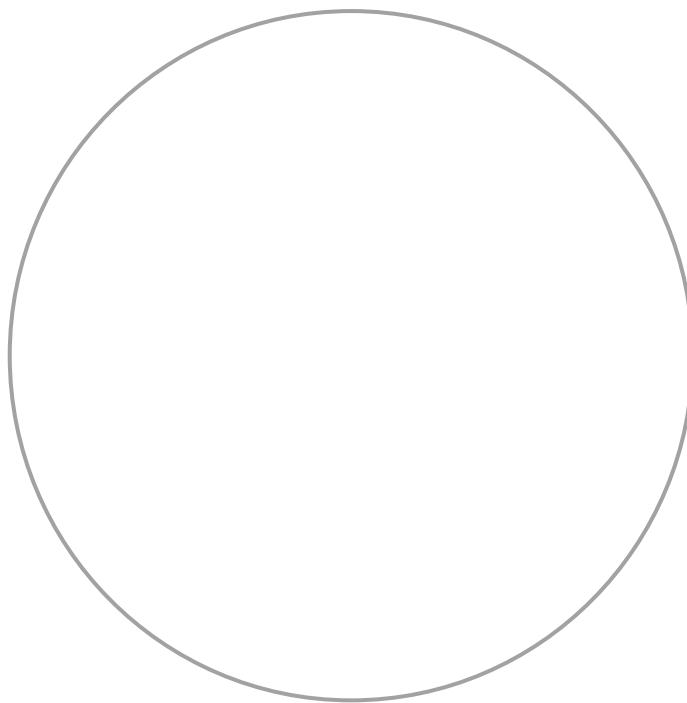
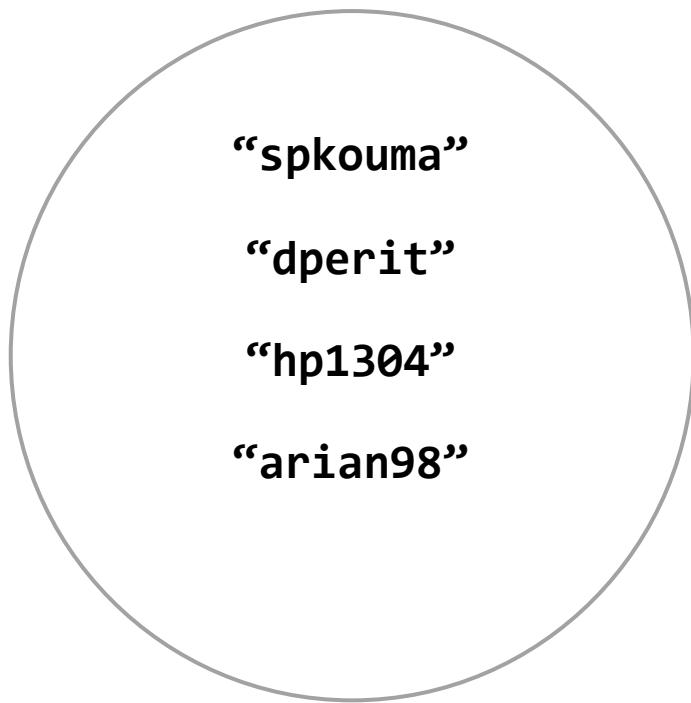
```
csAcct: [“spkouma”, “dperit”, “hp1304”, “arian98”,  
“spkouma”, “dperit”]
```



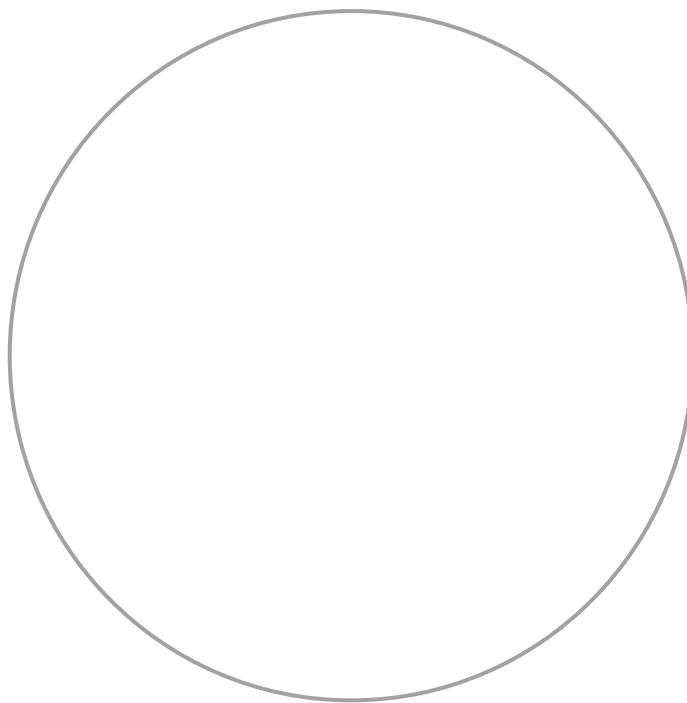
```
csAcct: [“spkouma”, “dperit”, “hp1304”, “arian98”,  
“spkouma”, “dperit”]
```



```
csAcct: [“spkouma”, “dperit”, “hp1304”, “arian98”,  
“spkouma”, “dperit”]
```



```
csAcct: [“spkouma”, “dperit”, “hp1304”, “arian98”,  
“spkouma”, “dperit”]
```



```
csAcct: [“spkouma”, “dperit”, “hp1304”, “arian98”,  
“spkouma”, “dperit”]
```

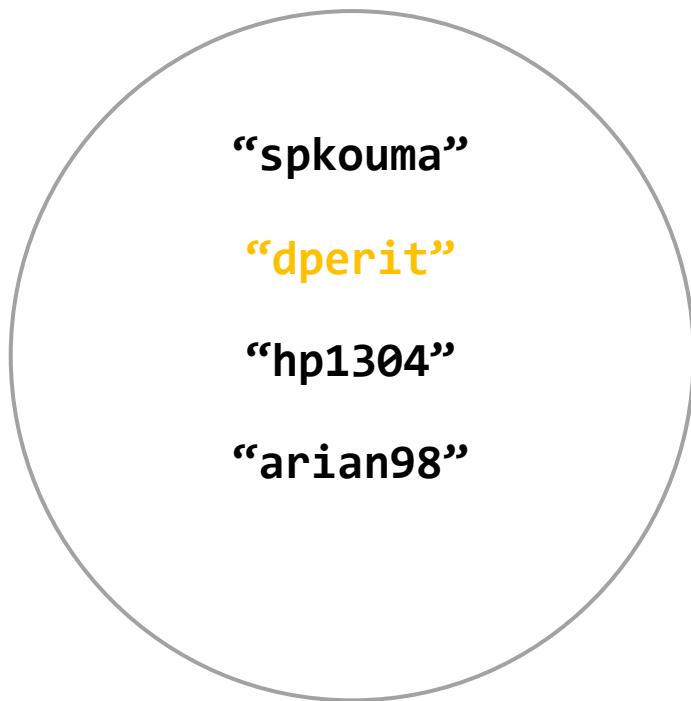


uniqueAcct



duplicatedAcct

```
csAcct: [“spkouma”, “dperit”, “hp1304”, “arian98”,  
“spkouma”, “dperit”]
```

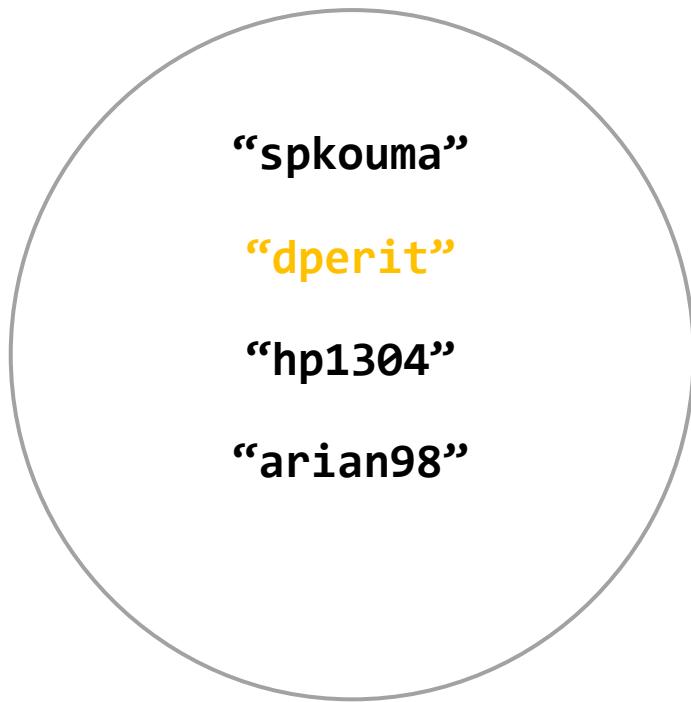


uniqueAcct



duplicatedAcct

```
csAcct: [“spkouma”, “dperit”, “hp1304”, “arian98”,  
“spkouma”, “dperit”]
```



uniqueAcct



duplicatedAcct

Set union

- ▶ suppose you have a set containing the accounts of students who completed assignment 1 and a second set containing the accounts of students who completed assignment 2
 - ▶ to find the set of students who completed either assignments, you compute the union of the two sets
 - ▶ **s1.addAll(s2)** : transforms **s1** into the union of **s1** and **s2**

```
// s1 contains the accounts who completed assignment 1  
// s2 contains the accounts who completed assignment 2
```

```
HashSet<String> eitherAssignment = new HashSet<String>(s1);  
eitherAssignment.addAll(s2);
```



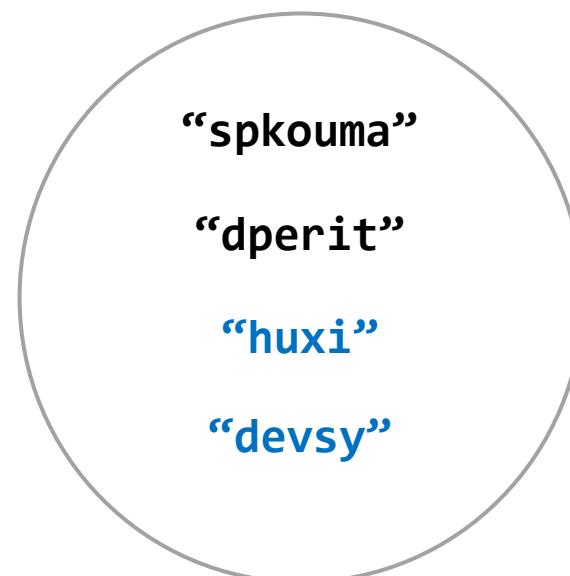
s1



s2



```
eitherAssignment = new HashSet<String>(s1);
```



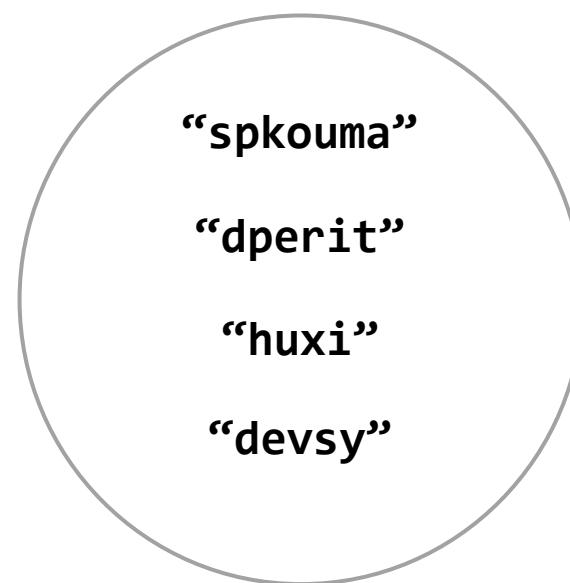
eitherAssignment.addAll(s2);

Set intersection

- ▶ suppose you have a set containing the accounts of students who completed assignment 1 and a second set containing the accounts of students who completed assignment 2
 - ▶ to find the set of students who completed both assignments, you compute the intersection of the two sets
 - ▶ **s1.retainAll(s2)** : transforms **s1** into the intersection of **s1** and **s2**

```
// s1 contains the accounts who completed assignment 1  
// s2 contains the accounts who completed assignment 2
```

```
HashSet<String> bothAssignment = new HashSet<String>(s1);  
bothAssignment.retainAll(s2);
```



```
bothAssignment = new HashSet<String>(s1);
```



s1



s2



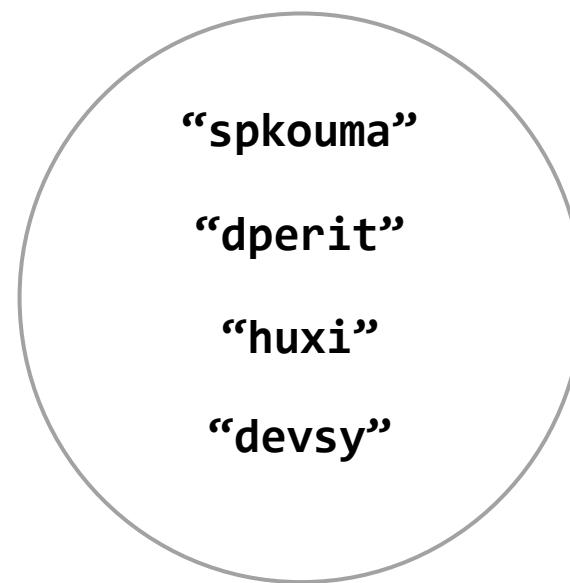
bothAssignment.retainAll(s2);

Set difference

- ▶ suppose you have a set containing the accounts of students who completed assignment 1 and a second set containing the accounts of students who completed assignment 2
 - ▶ to find the set of students who completed only assignment 1, you compute the set difference
 - ▶ **s1.removeAll(s2)** : transforms **s1** into the set difference of **s1** and **s2**

```
// s1 contains the accounts who completed assignment 1  
// s2 contains the accounts who completed assignment 2
```

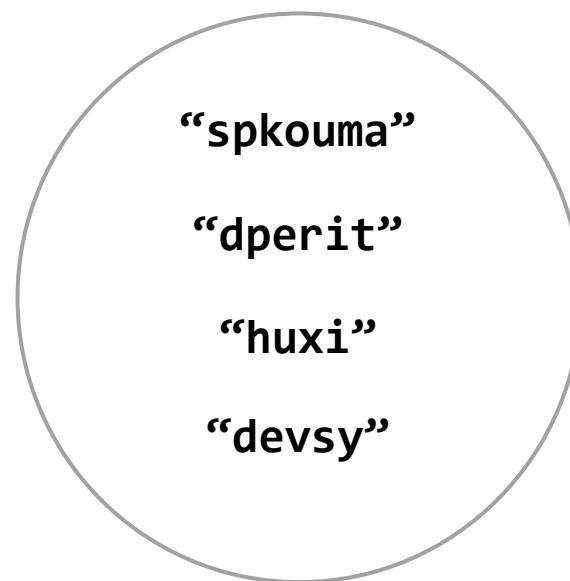
```
HashSet<String> onlyAssignment1 = new HashSet<String>(s1);  
onlyAssignment1.removeAll(s2);
```



```
onlyAssignment1 = new HashSet<String>(s1);
```



s1



s2



`onlyAssignment1.removeAll(s2);`

Maps

Maps

- ▶ map, dictionary, and associative array are all names for the same abstract data type
 - ▶ a *data type* is simply a type
 - ▶ an *abstract data type* is a (formal) specification of a type that may have many different implementations
 - ▶ the different implementations may have different advantages/disadvantages
- ▶ sets and lists are also abstract data types
- ▶ CISC235 is an entire course on data structures
 - ▶ a data structure is an implementation of a type that is intended for information storage, retrieval, and manipulation

Maps

- ▶ Java maps provide functionality similar to a Java list where the index can be any reference type
- ▶ also, Java allows the programmer to choose between several different kinds of maps
 - ▶ but there are two main kinds
 - ▶ **java.util.HashMap**
 - <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>
 - most similar to a Python dictionary for Python versions < 3.7
 - ▶ **java.util.TreeMap**
 - <https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html>
 - a dictionary that sorts its keys

Maps

- ▶ a third kind of map is also useful
- ▶ **java.util.LinkedHashMap**
 - <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html>
 - most similar to a Python dictionary for Python versions >= 3.7

Map interface

- ▶ **Map** models a group of elements that are accessed by keys
 - ▶ keys must be unique (i.e., the keys form a set)
 - ▶ each key maps to at most one value
 - ▶ multiple keys can map to the same value (i.e., the values form a collection that is not necessarily a set)
 - ▶ key-value pairs are called entries
- ▶ a **Map** is a 2-column table where the elements in the first column are unique
 - ▶ the entries are rows of the table

Map examples

	key	value
mathematical function $y = f(x)$	x	y
dictionary	word	list of definitions
number of days per month	month	number of days
provincial/territorial capitals	province	capital city
book index	keyword	set of pages
student marks in a course	student number	list of marks
student marks in a course	student number	map of names of marked items to marks

Map example

Key (month name)	Value (days)
“January”	31
“February”	28?
“March”	31
“April”	30
“May”	31
“June”	30
“July”	31
“August”	31
“September”	30
“October”	31
“November”	30
“December”	31

Creating a Map

- ▶ to create a map, you need a class that implements **Map**
- ▶ usually, you will want to use **HashMap**
 - ▶ very fast **get**, **put**, **containsKey**
 - $O(1)$ expected complexity
 - ▶ makes no guarantees as to the iteration order of the key set; in particular, it does not guarantee that the order will remain constant over time
- ▶ the other common choice is **TreeMap**
 - ▶ fast **get**, **put**, **containsKey** (but worse complexity than **HashMap**)
 - $O(\log n)$ complexity
 - ▶ keys are ordered by their natural ordering (sorted)
 - ▶ keys must have type that implements **Comparable** or the map must be initialized with a **Comparator** object

Creating a Map

- ▶ to create a map, you need a class that implements **Map**
- ▶ the other common choice is **LinkedHashMap**
 - ▶ very fast **get**, **put**, **containsKey** (but slower than **HashMap** and uses more memory)
 - $O(1)$ expected complexity
 - ▶ keys are maintained in insertion order

Adding entries to a map

- ▶ to add an entry to a map, use the **put** method
- ▶ the put method requires both a key and a value

```
TreeMap<String, String> capitals = new TreeMap<String, String>();  
  
capitals.put("Ontario", "Toronto");  
capitals.put("Quebec", "Quebec City");  
capitals.put("Nova Scotia", "Halifax");  
capitals.put("New Brunswick", "Fredericton");  
capitals.put("Manitoba", "Winnipeg");  
capitals.put("British Columbia", "Victoria");  
capitals.put("Prince Edward Island", "Charlottetown");  
capitals.put("Saskatchewan", "Regina");  
capitals.put("Alberta", "Edmonton");  
capitals.put("Newfoundland and Labrador", "St. John's");
```

Map example

Key (province name)	Value (city name)
Alberta	Edmonton
British Columbia	Vancouver
Manitoba	Winnipeg
Newfoundland and Labrador	St. John's
Nova Scotia	Halifax
Ontario	Toronto
Prince Edward Island	Charlottetown
Quebec	Quebec City
Saskatchewan	Regina

Getting a value from a map

- ▶ to get a value from a map, you need to provide a key
- ▶ the map uses the key to find the value associated with the key

```
String city = capitals.get("Ontario"); // city is "Toronto"
```

To determine if the map contains a key, use the **containsKey** method:

```
// capitals is the map of provinces to capital cities

if (!capitals.containsKey("Nunavut")) {
    capitals.put("Nunavut", "Iqaluit");
}

if (!capitals.containsKey("Northwest Territories")) {
    capitals.put("Northwest Territories", "Yellowknife");
}

if (!capitals.containsKey("Yukon")) {
    capitals.put("Yukon", "Whitehorse");
}
```

Counting the number of occurrences

- ▶ suppose that you have a list of strings and you want to know how many times each string occurs in the list
 - ▶ perhaps to find the most commonly occurring string
 - ▶ perhaps to find the least commonly occurring string
- ▶ you can use a map where the keys are the unique strings in the list and the values are the number of times each string occurs in the list

```
// t is a List of strings
HashMap<String, Integer> count = new HashMap<>();
for (String s : t) {
    if (count.containsKey(s)) {
        Integer n = count.get(s);      // int n also ok
        count.put(s, n + 1);
    }
    else {
        count.put(s, 1);
    }
}
```

keySet

- ▶ the keys in a map are unique so they form a set
- ▶ you can ask a map for its set of keys using the **keySet** method

```
// count is the map from the previous slide
```

```
Set<String> keys = count.keySet();
for (String k : keys) {
    Integer n = count.get(k);
    System.out.println(k + " occurs " + n + " times in t");
}
```

values

- ▶ the values in a map are not necessarily unique so they form a collection
- ▶ you can ask a map for its collection of values using the **values** method

```
// count is the map from two slides previous

Collection<Integer> vals = count.values();
int numStrings = 0;
for (Integer i : vals) {
    numStrings = numStrings + i;
}
double avgOccurrence = numStrings / count.keySet().size();
```

keyset and values

- ▶ notice that the **keySet** method does not specify what kind of set is returned:

```
Set<String> keys = count.keySet();
```

- ▶ similarly, the **values** method returns a type that we have not seen yet:

```
Collection<Integer> vals = count.values();
```

- ▶ both methods return a reference to an *interface*

Interfaces

- ▶ in its most common form, a Java interface is a specification (but *not* an implementation) of an API
 - ▶ the interface says what methods exist and what the methods promise to do, but it does not define how the methods are implemented
 - ▶ e.g., the **List** interface specifies what methods every list must provide
- ▶ a class that implements the interface must provide an implementation of every method in the interface
 - ▶ e.g., **ArrayList** implements every method in the List interface

Interfaces are types

- ▶ an interface is a reference data type
 - ▶ this means that you can make a variable or parameter whose type is an interface
 - ▶ <https://docs.oracle.com/javase/tutorial/java/IandI/interfaceAsType.html>

```
List<String> t = new ArrayList<String>();
```

interface

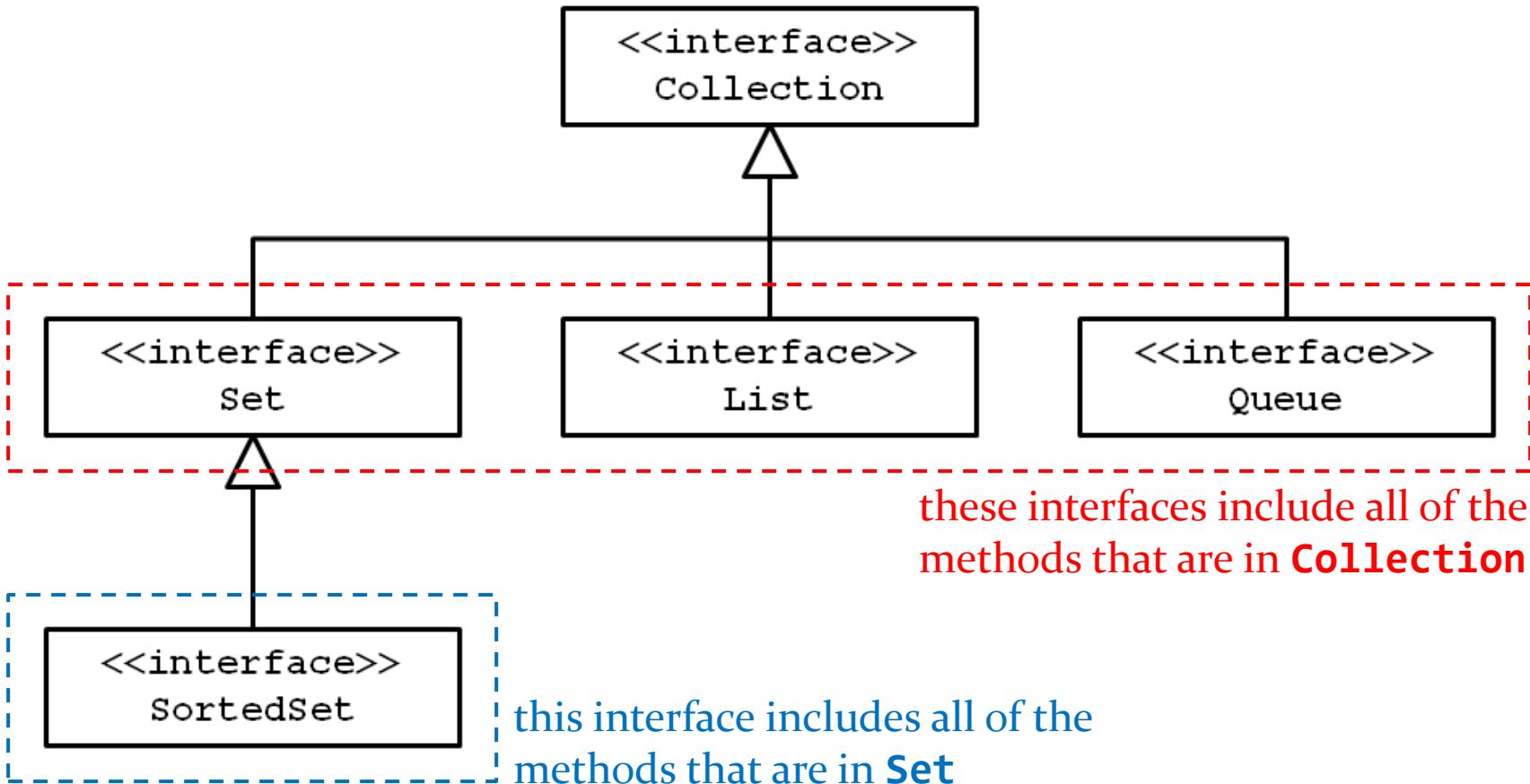
implements the interface

```
Map<String, Integer> count = new HashMap<>();
```

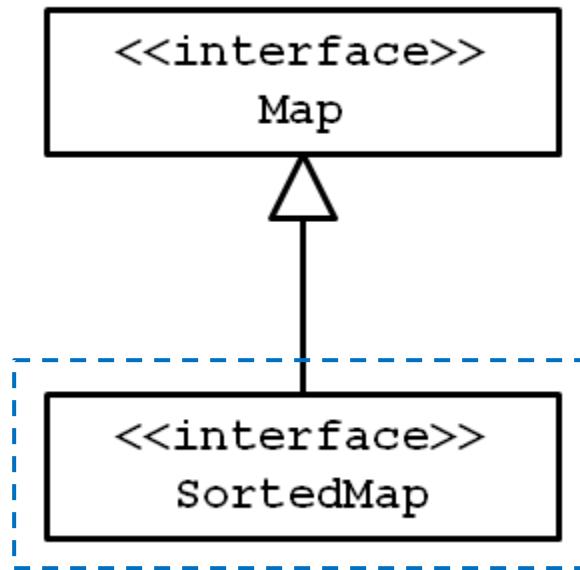
Java Collection Framework

- ▶ made up of:
 - ▶ interfaces
 - ▶ these define what methods the various types of collections support
 - ▶ classes
 - ▶ these implement the interfaces
 - ▶ algorithms
 - ▶ these are the methods that operate on collections (such as sorting a collection and searching a collection)

Collection hierarchy UML class diagram



Map hierarchy UML class diagram



this interface includes all of the methods that are in `Map`

Collection interface

- ▶ a **Collection** represents a group of objects where each object is called an element of the collection
- ▶ the interface defines the most general operations that a user can ask a collection to perform

```
public interface Collection<E> extends Iterable<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);                                //optional  
    boolean remove(Object element);                        //optional  
    Iterator<E> iterator();  
  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);          //optional  
    boolean removeAll(Collection<?> c);                 //optional  
    boolean retainAll(Collection<?> c);                //optional  
    void clear();                                         //optional  
  
    // Array operations  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```

List interface

- ▶ a **List** is a **Collection** that holds its elements in numbered sequence. **List** supports:
 - ▶ positional access/mutation
 - ▶ manipulates elements based on their numerical position in the list
 - ▶ search
 - ▶ searches for a specified object in the list and returns its numerical position
 - ▶ iteration
 - ▶ extends Iterator semantics to take advantage of the list's sequential nature
 - ▶ range-view
 - ▶ performs arbitrary range operations on the list

```
public interface List<E> extends Collection<E> {  
    // Positional access  
    E get(int index);  
    E set(int index, E element);          //optional  
    boolean add(E element);              //optional  
    void add(int index, E element);      //optional  
    E remove(int index);                //optional  
    boolean addAll(int index,  
                  Collection<? extends E> c );      //optional  
  
    // Search  
    boolean contains(Object o);  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    // Iteration  
    ListIterator<E> listIterator();  
    ListIterator<E> listIterator(int index);  
  
    // Range-view  
    List<E> subList(int from, int to);  
}
```

Set interface

- ▶ a **Set** is a **Collection** that cannot contain duplicate elements
- ▶ it models the mathematical set abstraction
 - ▶ supports set union, intersection, and difference
- ▶ the **Set** interface contains only methods inherited from **Collection** and adds the restriction that duplicate elements are prohibited

Map interface

- ▶ **Map** does not extend **Collection**; instead, it is a separate interface

```
public interface Map<K,V> {  
    // Basic operations  
    V put(K key, V value);  
    V get(Object key);  
    V remove(Object key);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
  
    // Bulk operations  
    void putAll(Map<? extends K, ? extends V> m);  
    void clear();
```

```
// Collection Views  
public Set<K> keySet();  
public Collection<V> values();  
public Set<Map.Entry<K,V>> entrySet();  
  
// Interface for entrySet elements  
public interface Entry {  
    K getKey();  
    V getValue();  
    V setValue(V value);  
}  
}
```

Fundamental Terms

Type

- ▶ a type is a set of values and the operations that can be performed with those values
- ▶ for example:
- ▶ **int**
 - ▶ 32-bit signed integer value between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`
 - ▶ `+, -, *, /, %, ==, !=, >, <`, and more

Type

- ▶ a type is a set of values and the operations that can be performed with those values
- ▶ for example:
 - ▶ **java.lang.String**
 - ▶ a sequence of zero or more Unicode characters
 - ▶ `==`, `equals`, `charAt`, `indexOf`, `substring`, and many more

Type

- ▶ a type is a set of values and the operations that can be performed with those values
- ▶ for example:
 - ▶ **java.util.List**
 - ▶ a sequence of zero or more elements of the same type
 - ▶ `==`, `equals`, `get`, `set`, `subList`, and many more

Primitive Types

- ▶ in Java the primitive types are the built-in numeric types and the type boolean

Category	Type
Integer	byte
	char
	short
	int
	long
Floating-point	float
	double
True/false	boolean

Reference Types

- ▶ every type that is not primitive is a reference type
- ▶ for example:
 - ▶ **int[]**
 - ▶ an array of int values
 - ▶ all arrays (even arrays of primitive values) are reference type
 - ▶ **java.lang.String**
 - ▶ **java.util.List**
 - ▶ **java.util.ArrayList**
 - ▶ and so on...

Class

- ▶ a class is an implementation of a reference type
 - ▶ in Java, a class is also a type but not all types are classes
- ▶ for example:
 - ▶ **java.lang.String**
 - ▶ **java.util.ArrayList**
 - ▶ an implementation of the type **java.util.List**
 - ▶ **java.util.List** is not a class

Object

- ▶ an object is an instance of a class
 - ▶ a class is a blueprint that is used to make objects
- ▶ for example:

```
List<String> t = new ArrayList<>();
```



uses the **ArrayList** class to
create an **ArrayList** object

Reference

- ▶ a reference is an identifier that can be used to find a particular object
- ▶ in many Java implementations, references are memory addresses
- ▶ in Java, any variable whose type is not a primitive type stores a reference
- ▶ for example:

```
List<String> t = new ArrayList<>();  
        ↴
```

t is a reference to the object created on
the right-hand side of the assignment

Memory Diagrams

- ▶ a memory diagram can be helpful for understanding the differences between primitive and reference types
- ▶ a memory diagram is simply a table with 3 columns

variable name	address	value
	98	
	99	
	100	
	101	
	102	
	...	
	700	

Memory Diagrams

```
double x = -1.0;
```

```
List<String> t = new ArrayList<>();
```

variable name	address	value
	98	
	99	
x	100	-1.0
t	101	700a
	102	
	...	
	700	ArrayList<String> object

*I use an a to indicate a value is an address instead of a primitive numeric value:
700a means “the address 700”

The object model

The object model

- ▶ procedural programming focuses on decomposing programs into functions that are passed data to operate on
- ▶ object-oriented programming focuses on decomposing programs into interacting objects that contain data and can operate on their own data
- ▶ four main elements
 1. abstraction
 2. encapsulation
 3. modularity
 4. hierarchy (covered later in the course)

Abstraction

- ▶ "Abstraction is one of the fundamental ways that we as humans cope with complexity... An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer."
- ▶ [Booch, G. et al. 2007. Object-Oriented Analysis and Design with Applications (3rd Edition)]

Encapsulation

- ▶ "Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; **encapsulation serves to separate the contractual interface of an abstraction and its implementation.**"
- ▶ [Booch, G. et al. 2007. Object-Oriented Analysis and Design with Applications (3rd Edition)]

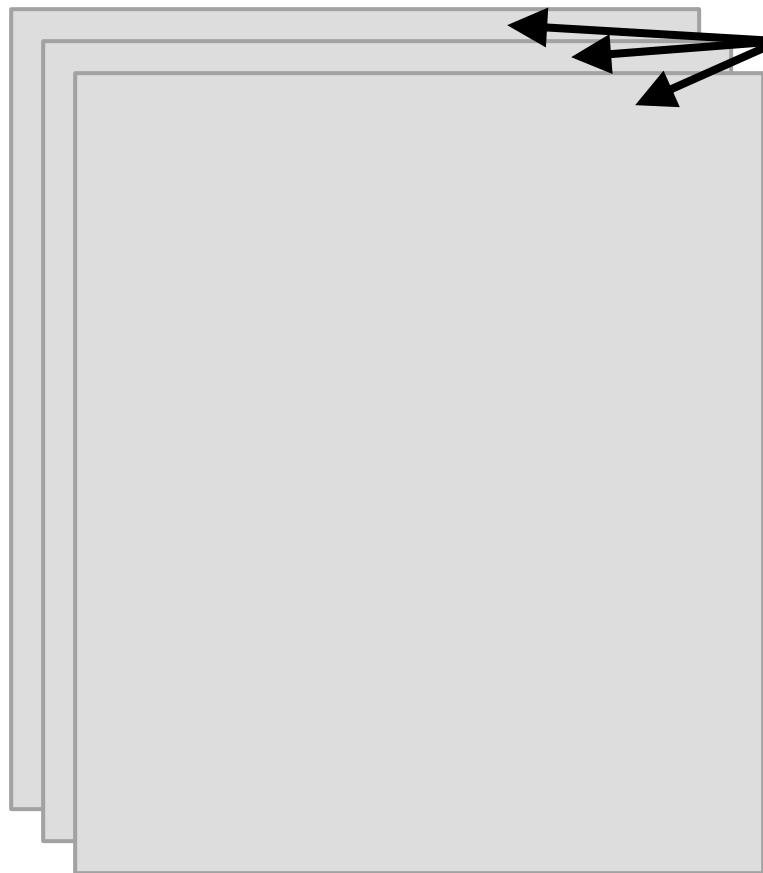
Modularity

- ▶ "The act of **partitioning a program into individual components** can reduce its complexity to some degree.... Although partitioning a program is helpful for this reason, a more powerful justification for partitioning a program is that it **creates a number of well-defined, documented boundaries within the program**. These boundaries, or interfaces, are invaluable in the comprehension of the program."
- ▶ [Myers, G. 1978. Composite/Structured Design][Quoted by Booch, G. et al. 2007. Object-Oriented Analysis and Design with Applications (3rd Edition)]

Organization of a Java Program

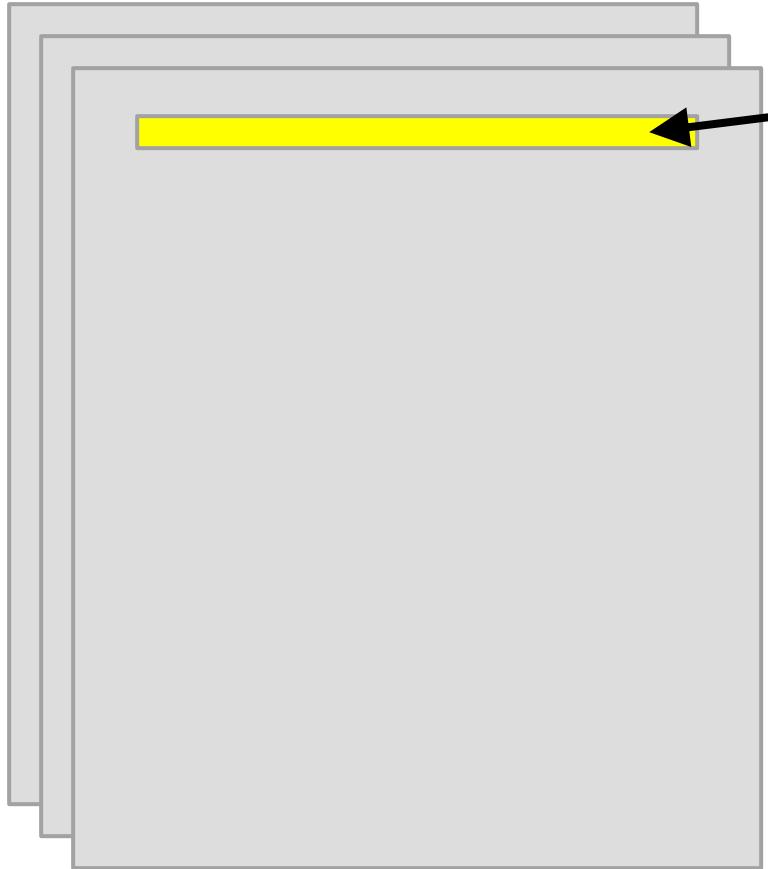
Packages, classes, fields, and methods

Organization of a Typical Java Program



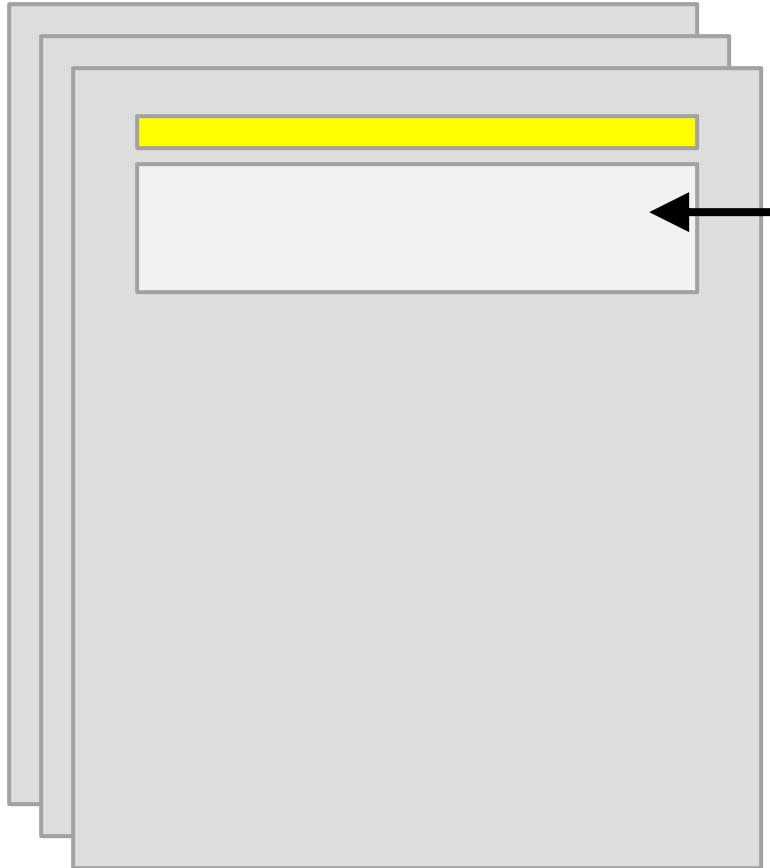
▶ one or more files

Organization of a Typical Java Program



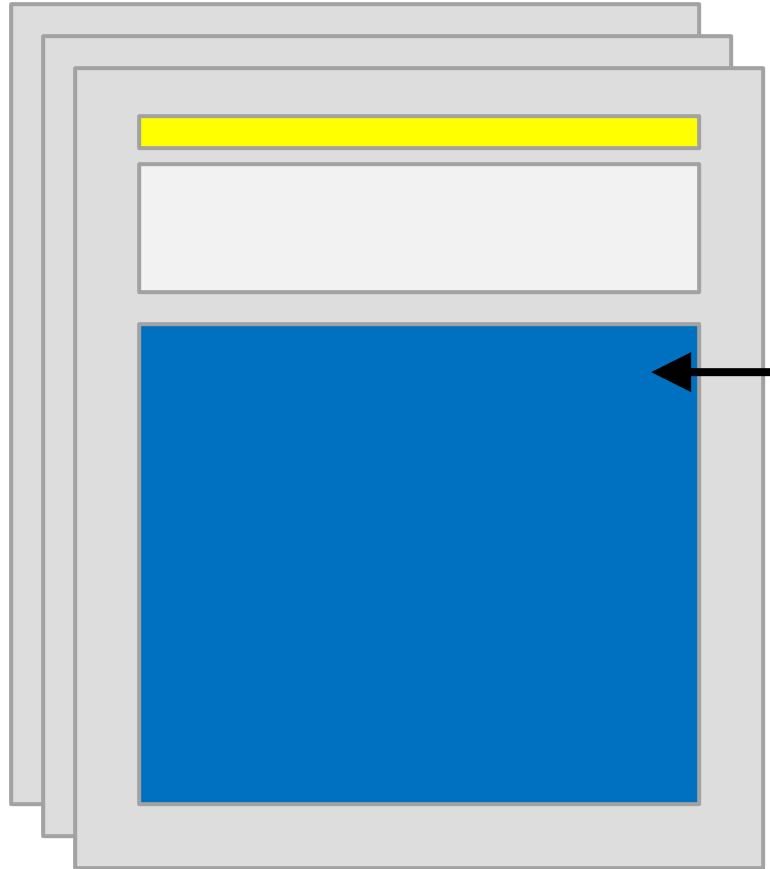
- ▶ one or more files
- ▶ zero or one package name

Organization of a Typical Java Program



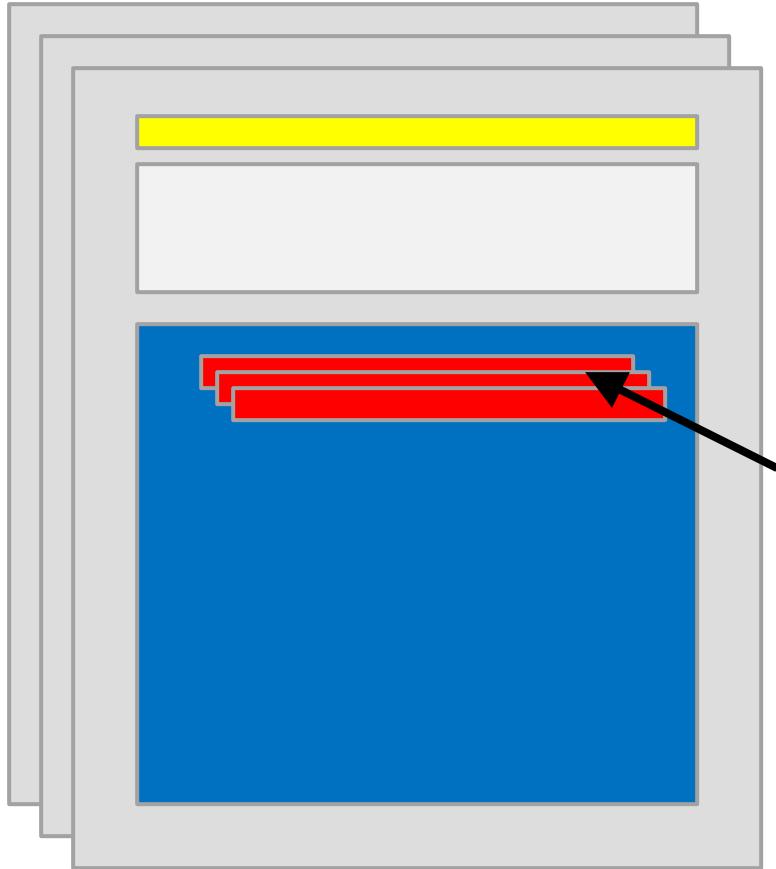
- ▶ one or more files
- ▶ zero or one package name
- ▶ zero or more import statements

Organization of a Typical Java Program



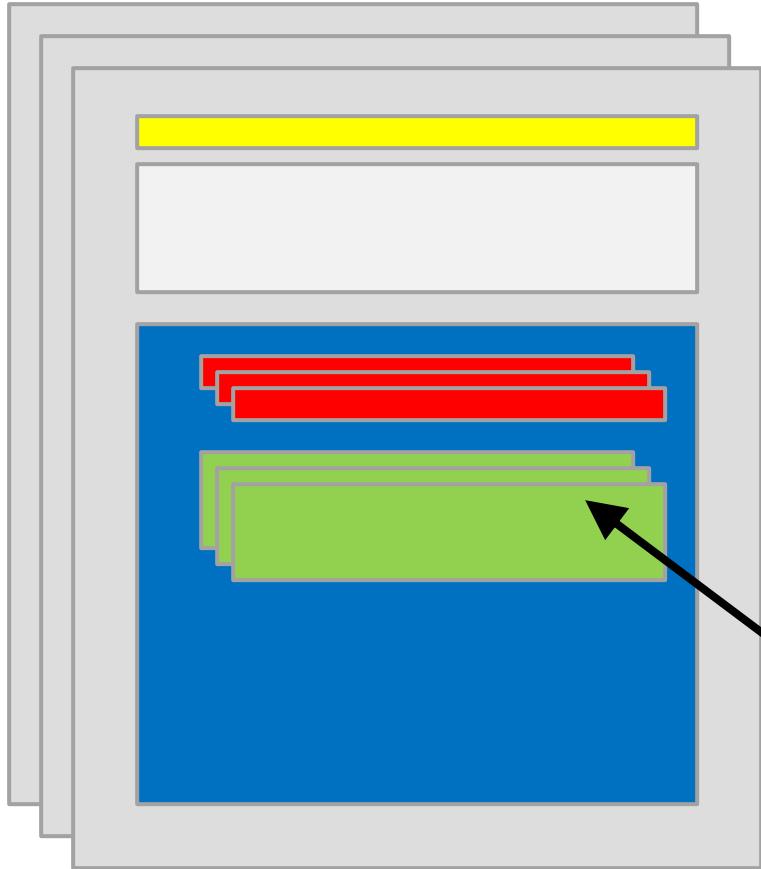
- ▶ one or more files
- ▶ zero or one package name
- ▶ zero or more import statements
- ▶ **one class**

Organization of a Typical Java Program



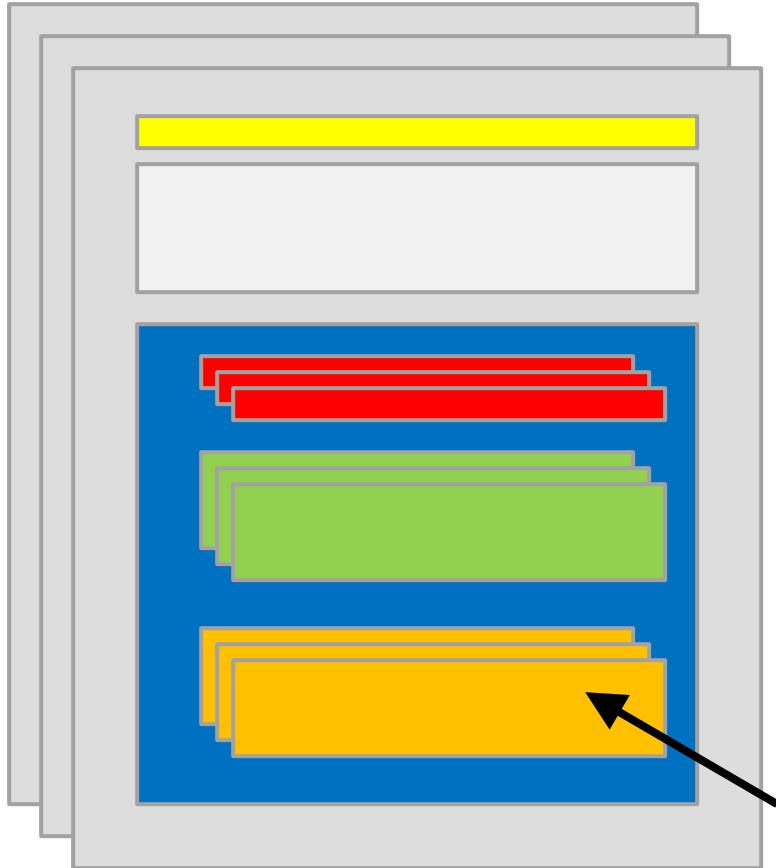
- ▶ one or more files
- ▶ zero or one package name
- ▶ zero or more import statements
- ▶ one class
- ▶ one or more fields (class variables)

Organization of a Typical Java Program



- ▶ one or more files
- ▶ zero or one package name
- ▶ zero or more import statements
- ▶ one class
- ▶ zero or more fields (class variables)
- ▶ zero or more constructors

Organization of a Typical Java Program



- ▶ one or more files
- ▶ zero or one package name
- ▶ zero or more import statements
- ▶ one class
- ▶ zero or more fields (class variables)
- ▶ zero or more constructors
- ▶ zero or more methods

```
zero or one package declaration •• package ca.queensu.cs.cisc124.notes.basics;
zero or more import statements •• import java.util.ArrayList;
one or more class declarations •• public class Stack {

    private ArrayList<String> stack; •• zero or more fields

    public Stack() {
        this.stack = new ArrayList<>();
    } •• zero or more constructors

    public int size() {
        return this.stack.size();
    }

    public void push(String elem) {
        this.stack.add(elem);
    }

    public String pop() { •• zero or more methods
        String elem = this.stack.remove(this.size() - 1);
        return elem;
    }

    public String toString() {
        StringBuilder b = new StringBuilder("ListStack:");
        if (this.size() != 0) {
            for (int i = this.size() - 1; i >= 0; i--) {
                b.append('\n');
                b.append(this.stack.get(i));
            }
        }
        return b.toString();
    }
}
```

Organization of a Typical Java Program

- ▶ it's actually more complicated than this
 - ▶ static initialization blocks
 - ▶ non-static initialization blocks
 - ▶ classes inside of classes (inside of classes ...)
 - ▶ classes inside of methods
 - ▶ anonymous classes
 - ▶ lambda expressions (in Java 8)
 - ▶ modules (in Java 9)

- ▶ see <http://docs.oracle.com/javase/tutorial/java/javaOO/index.html>

Packages

- ▶ packages are used to organize Java classes into namespaces
- ▶ a namespace is a container for names
 - ▶ the namespace also has a name

Packages

- ▶ packages are used to organize related classes and interfaces
- ▶ e.g., all of the Java API classes are in the package named **java**

Packages

- ▶ packages can contain subpackages
 - ▶ e.g., the package **java** contains packages named **lang**, **util**, **io**, etc.
- ▶ the fully qualified name of the subpackage is the fully qualified name of the parent package followed by a period followed by the subpackage name
 - ▶ e.g., **java.lang**, **java.util**, **java.io**

Packages

- ▶ packages can contain classes and interfaces
 - ▶ e.g., the package **java.lang** contains the classes **Object**, **String**, **Math**, etc.
- ▶ the fully qualified name of the class is the fully qualified name of the containing package followed by a period followed by the class name
 - ▶ e.g., **java.lang.Object**, **java.lang.String**, **java.lang.Math**

Packages

- ▶ packages are supposed to ensure that fully qualified names are unique
- ▶ this allows the compiler to disambiguate classes with the same unqualified name, e.g.,

```
your.String s = new your.String("hello");
String t = "hello";
```

Packages

- ▶ how do we ensure that fully qualified names are unique?
- ▶ package naming convention
 - ▶ packages should be organized using your domain name in reverse, e.g.,
 - ▶ CS domain name **cs.queensu.ca**
 - ▶ package name **ca.queensu.cs**

Classes

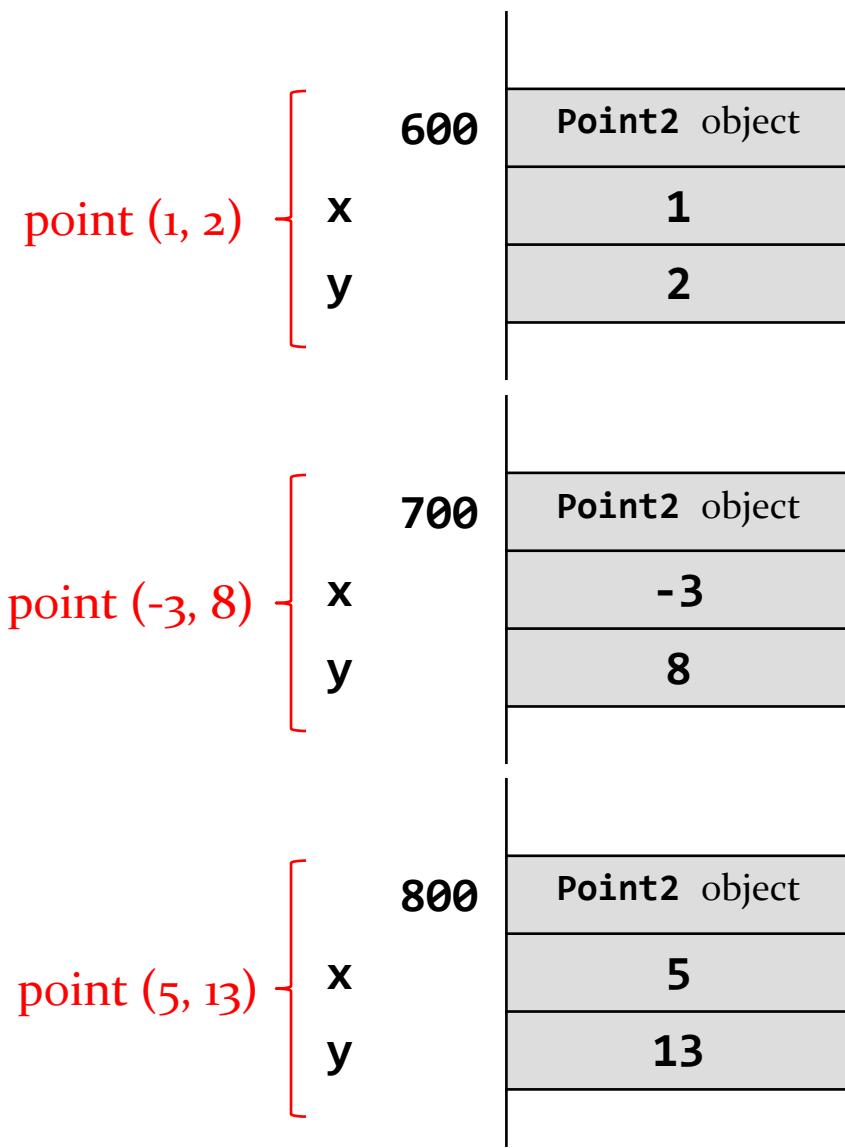
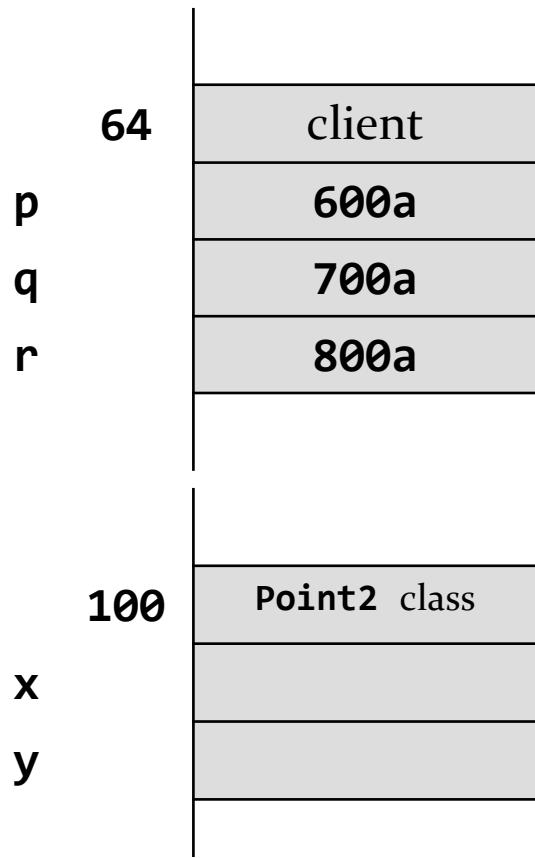
Classes

- ▶ a class is an implementation of a type
- ▶ a class is (usually) used as a blueprint to make instances of the class (objects)

Why objects?

- ▶ each object has its own copy of all non-static fields
 - ▶ this allows objects to have their own *state*
 - ▶ in Java the state of an object is the set of current values of all of its non-static fields
 - ▶ e.g., we can create multiple **Point2** objects that all represent different two-dimensional points

```
Point2 p = new Point2(1, 2);  
Point2 q = new Point2(-3, 8);  
Point2 r = new Point2(5, 13);
```



Implementing classes

- ▶ many classes represent kinds of values
 - ▶ examples of values: name, date, colour, mathematical point or vector
 - ▶ Java examples: **String**, **Date**, **Integer**
- ▶ when implementing a value class you need to decide what data each object needs to have
 - ▶ in other words, you need to decide which variables you need to represent the state of each object
- ▶ the variables that represent the state of an object are called *fields*

Implementing a point class

- ▶ consider implementing a class that represents 2-dimensional points
- ▶ what fields do you need to represent a point?
 - ▶ a possible implementation would have:
 - ▶ a field to represent the x-coordinate of the point
 - ▶ a field to represent the y-coordinate of the point

```
package ca.queensu.cs.cisc124.notes.basics.geometry;
```

```
/**  
 * A Cartesian point in 2-dimensions having real  
 * coordinates.  
 */
```

```
public class Point2 {  
    /**  
     * The coordinates of this point.  
     */  
    private double x;  
    private double y;  
}
```

public class: any client can use this class

private fields: only the class can use these fields

Top level classes

- ▶ a top level class is a class that is not nested inside of another class
 - ▶ **Point2** is a top level class
-
- ▶ a top level class can have either:
 - ▶ no access modifier
 - ▶ called package private access; the class is visible only inside its own package (and not its subpackages)
 - ▶ the **public** access modifier
 - ▶ the class is visible to all other classes everywhere

Access modifiers on fields

- ▶ fields can have one of four different access modifiers:
 - ▶ **private**
 - ▶ field is visible only to the class that the field is in
 - ▶ no access modifier (package private)
 - ▶ same as private and field is also visible to all other classes in the same package
 - ▶ **protected**
 - ▶ same as package private and field is also visible to all subclasses of the class that the field is in
 - ▶ **public**
 - ▶ field is visible to all classes everywhere

Access modifiers on fields

- ▶ the safe rules of thumb:
 - ▶ a field that represents a constant value can be **public**
 - ▶ e.g., PI in `java.lang.Math`
 - ▶ all other fields should be **private**
- ▶ **protected** fields appear when using inheritance

Implementing a counter class

- ▶ consider implementing a class that represents a counter that counts up starting from zero
- ▶ what fields do you need to represent a point?
 - ▶ a possible implementation would have:
 - ▶ a field to represent the count

```
package ca.queensu.cs.cisc124.notes.basics.counter;

/**
 * The {@code Counter} class represents a device used to
 * incrementally count upwards from zero up to
 * {@code Integer.MAX_VALUE}.
 *
 */
public class Counter {

    /**
     * The current value of this counter.
     */
    private int value;

}

}
```

Implementing a stopwatch class

- ▶ consider implementing a class that represents a stopwatch that the user can start and stop
 - ▶ reports the time elapsed between starting and stopping
- ▶ what fields do you need to represent a point?
 - ▶ a possible implementation would have:
 - ▶ ???

Implementing a stopwatch class

- ▶ to get the current time you can use the method
System.nanoTime
- ▶ **nanoTime** gives you the most precise available system timer, in nanoseconds, relative to some fixed but arbitrary time
- ▶ the elapsed time for the stopwatch will be proportional to the difference (*stop_time* – *start_time*)
- ▶ "proportional to" because we will report the elapsed time in seconds instead of nanoseconds

```
package ca.queensu.cs.cisc124.notes.basics.stopwatch;  
/**  
 * A simple stopwatch for recording elapsed time.  
 *  
 */  
public class Stopwatch {  
  
    /**  
     * Raw time (from nanoTime) when the stopwatch is started  
     */  
    private long rawStartTime;  
  
    /**  
     * Raw time (from nanoTime) when the stopwatch is stopped  
     */  
    private long rawStopTime;  
  
    /**  
     * True if stopwatch is started and not stopped, false  
     * otherwise  
     */  
    private boolean isRunning;  
}
```

Using our classes

- ▶ even in its current form, we can use our classes to create objects
- ▶ however, we can't do much with the created objects

```
package lectures.basics;

import ca.queensu.cs.cisc124.notes.basics.counter;
import ca.queensu.cs.cisc124.notes..basics.geometry.Point2;
import ca.queensu.cs.cisc124.notes.basics.stopwatch.Stopwatch;

public class MakeSomeObjects {

    public static void main(String[] args) {
        // create a point
        Point2 p = new Point2();

        // create a counter
        Counter c = new Counter();

        // create a stopwatch
        Stopwatch w = new Stopwatch();

    }
}
```

Encapsulation

- ▶ we can add features to our classes to make them more useful
- ▶ e.g., we can add methods that *use the fields* of **Stopwatch** to perform some sort of computation (like compute the elapsed time since the watch was started)
- ▶ e.g., we can add constructors that *set the values of the fields* of a **Point2** object when it is created
- ▶ in object oriented programming the term *encapsulation* means bundling data and methods that use the data into a single unit

Constructors

- ▶ the purpose of a constructor is to initialize the state of an object
- ▶ *it should set the values of all of the non-static fields to appropriate values*
- ▶ a constructor:
 - ▶ must have the same name as the class
 - ▶ never returns a value (not even **void**)
 - ▶ constructors are not methods
 - ▶ can have zero or more parameters

Classes with no constructors

- ▶ a constructor must be called whenever a new object is created
 - ▶ but our classes did not define any constructor!
-
- ▶ if a class defines no constructors, then the Java compiler will silently insert a no-argument constructor

No-argument constructor

- ▶ the no-argument constructor has zero parameters
- ▶ the no-argument constructor initializes the state of an object to some well defined state chosen by the implementer
- ▶ the compiler generated no-argument constructor initializes
 - ▶ primitive-type numeric fields to zero
 - ▶ primitive-type boolean fields to false, and
 - ▶ reference-type fields to **null**

Point2 no-argument constructor

- ▶ initializes the point to have coordinates (0, 0)

```
package ca.queensu.cs.cisc124.notes.basics.geometry;

public class Point2 {
    private double x;
    private double y;

    /**
     * Initializes the coordinates of this point to
     * {@code (0.0, 0.0)}.
     */

    public Point2() {
        this.x = 0.0;
        this.y = 0.0;
    }
}
```

Inside a constructor, the keyword **this** is a reference to the object that is currently being initialized.

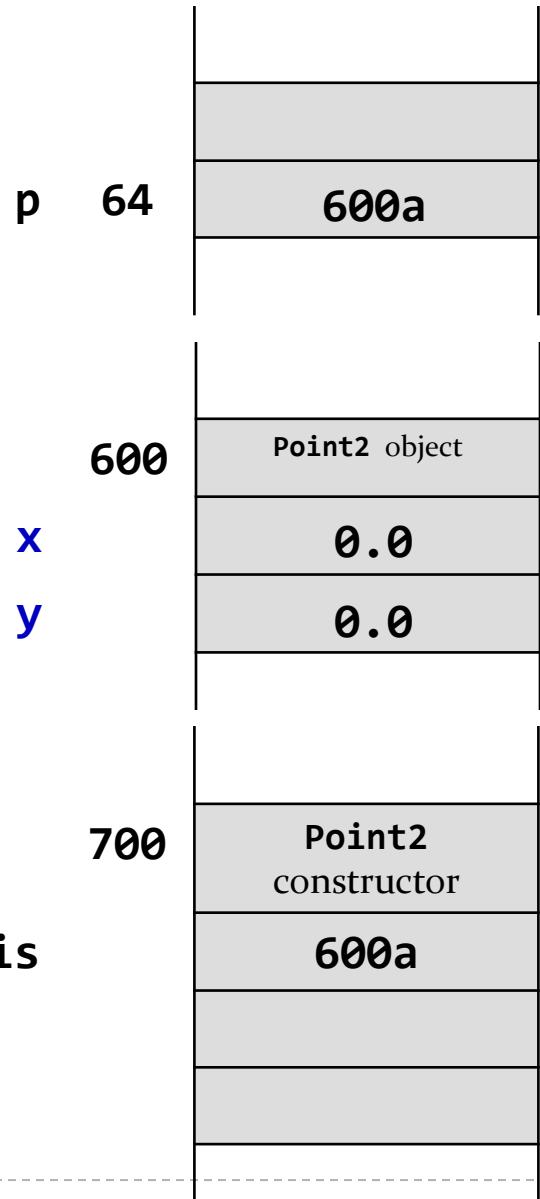
A constructor can have an access modifier; the possible modifiers are identical to the access modifiers for fields.

```
Point2 p = new Point2();
```

1. a reference variable **p** is created somewhere in memory
2. **new** allocates memory for a **Point2** object
3. the **Point2** constructor is invoked by passing the memory address of the object
4. the constructor runs, setting the values of the fields **this.x** and **this.y**
5. the value of **p** is set to the memory address of the constructed object

fields {

this



Counter no-argument constructor

- ▶ initializes the counter value to be 0

```
package ca.queensu.cs.cisc124.notes.basics.counter;

public class Counter {

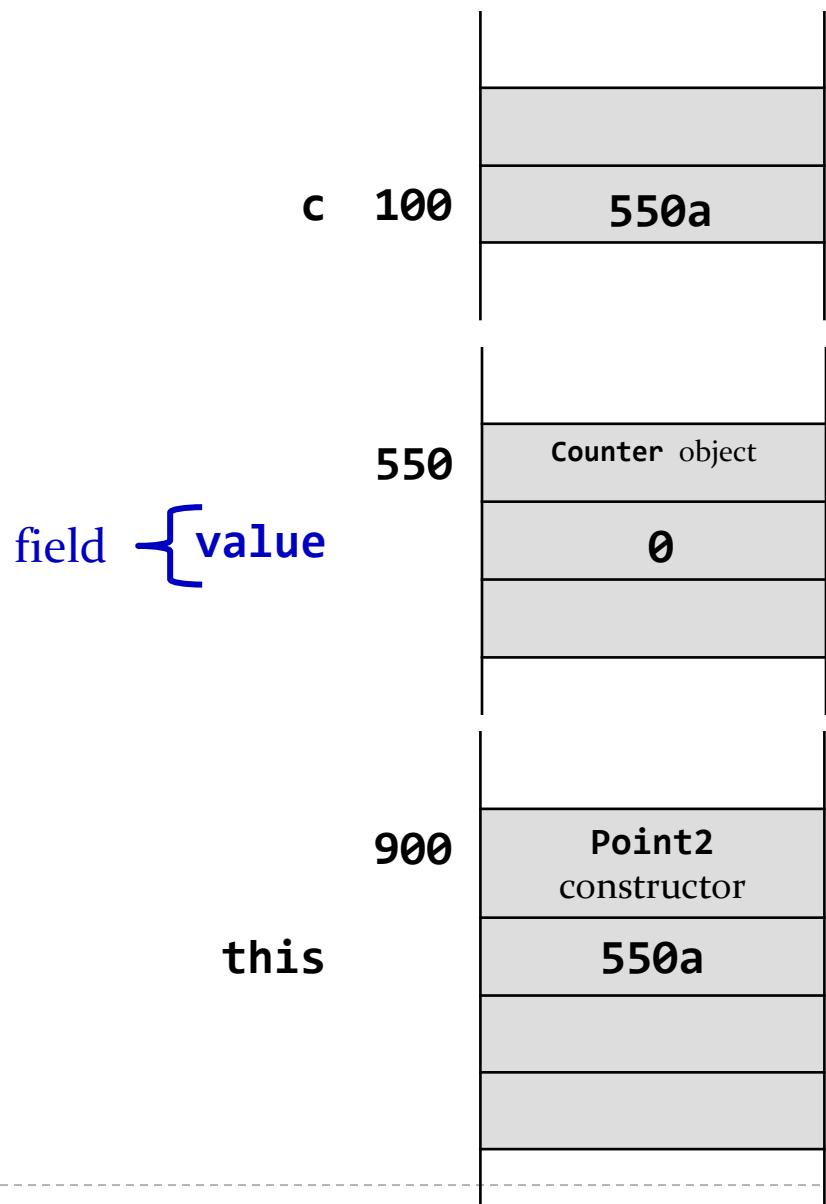
    private int value;

    /**
     * Initializes this counter so that its current value is 0.
     */
    public Counter() {
        this.value = 0;
    }

}
```

```
Counter c = new Counter();
```

1. a reference variable **c** is created somewhere in memory
2. **new** allocates memory for a **Counter** object
3. the **Counter** constructor is invoked by passing the memory address of the object
4. the constructor runs, setting the values of the fields **this.value**
5. the value of **c** is set to the memory address of the constructed object



Stopwatch no-argument constructor

- ▶ initializes the
 - ▶ start time to be 0
 - ▶ stop time to be 0
 - ▶ the stopwatch to be not running

```
package ca.queensu.cs.cisc124.notes.basics.stopwatch;

public class Stopwatch {

    private long rawStartTime;
    private long rawStopTime;
    private boolean isRunning;

    /**
     * Initializes this stopwatch so that the elapsed time
     * is zero seconds.
     */
    public Stopwatch() {
        this.rawStartTime = 0L;
        this.rawStopTime = 0L;
        this.isRunning = false;
    }

}
```

Methods

- ▶ a method performs some kind of computation
- ▶ a *non-static* method can use any field belonging to an object in the computation
- ▶ for example, we can provide a non-static method that allows the client to set both the x and y coordinates of the point

```
/**  
 * Sets the x and y coordinate to the specified values.  
 *  
 * @param newX the new x coordinate  
 * @param newY the new y coordinate  
 * @return a reference to this point  
 */  
public Point2 set(double newX, double newY) {  
    this.x = newX;  
    this.y = newY;  
    return this;  
}
```

Methods

- ▶ we can provide a method that allows the client to set just the x coordinate of the point

```
/**  
 * Sets the x coordinate to the specified value.  
 *  
 * @param newX the new x coordinate  
 * @return a reference to this point  
 */  
public Point2 x(double newX) {  
    this.x = newX;  
    return this;  
}
```

Methods

- ▶ we can provide a method that allows the client to set just the y coordinate of the point

```
/**  
 * Sets the y coordinate to the specified value.  
 *  
 * @param newY the new y coordinate  
 * @return a reference to this point  
 */  
public Point2 y(double newY) {  
    this.y = newY;  
    return this;  
}
```

Mutator methods

- ▶ methods that change the state of an object are called *mutator methods*
- ▶ **set(double, double)**, **x(double)**, and **y(double)** are all mutator methods

Accessor methods

- ▶ a method that returns information about the state of an object is called an *accessor method*
- ▶ for example, we can provide methods that return the x or y coordinate of a point

```
/**  
 * Returns the x coordinate of this point.  
 *  
 * @return the x coordinate of this point  
 */  
public double x() {  
    return this.x;  
}
```

```
/**  
 * Returns the y coordinate of this point.  
 *  
 * @return the y coordinate of this point  
 */  
public double y() {  
    return this.y;  
}
```

Method header

- ▶ the first line of a method declaration is sometimes called the *method header*
- ▶ a method header is made up of the following parts:
 1. modifiers
 2. return type
 3. method name
 4. parameter list
 5. throws clause (only if the method throws a checked exception)

Method header

```
public Point2 set(double x, double y)
```



modifier

Method header

```
public Point2 set(double x, double y)
```



return type

Method header

```
public Point2 set(double x, double y)
```



method name

Method header

```
public Point2 set(double x, double y)
```



parameter list

Parameter list

- ▶ the parameter list is the list of types and names that appear inside of the parentheses

```
public Point2 set(double x, double y)
```

parameter list

- ▶ the names in the parameter list must be unique
 - ▶ i.e., duplicate parameter names are not allowed

Method signature

- ▶ every method has a *signature*
 - ▶ the signature consists of the method name and the types in the parameter list

```
public Point2 set(double x, double y)
```

has the signature

```
set(double, double)
```

Method signature

- ▶ other examples from **java.lang.String**
 - ▶ headers
 - ▶ **String toUpperCase()**
 - ▶ **char charAt(int index)**
 - ▶ **int indexOf(String str, int fromIndex)**
 - ▶ **void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)**
 - ▶ signatures
 - ▶ **toUpperCase()**
 - ▶ **charAt(int)**
 - ▶ **indexOf(String, int)**
 - ▶ **getChars(int, int, char[], int)**

Method signature

- ▶ method signatures in a class must be unique
- ▶ we can introduce a second method in the same class:

```
public Point2 set(double x) // (x, 0)
```

- ▶ but not a third one:

```
public Point2 set(double y) // (0, y)
```

Method signature

- ▶ two or more methods with the same name but different signatures are said to be *overloaded*

```
public double distanceTo(float x, float y)
```

```
public double distanceTo(double x, double y)
```

```
public double distanceTo(Point2 p)
```

are all overloaded methods

Method return types

- ▶ all Java methods return nothing (**void**) or a single type of value
- ▶ our method

```
public double x()
```

has the return type **double**

Methods common to all classes

- ▶ in Java every class is actually a child class of the class **java.lang.Object**
- ▶ this means that every class has methods that it inherits from **java.lang.Object**
 - ▶ there are 11 such methods, but 3 are especially important to us:
 - **toString**
 - **equals**
 - **hashCode**

toString

- ▶ the **toString** method should return a textual representation of the object
- ▶ a textual representation of the point **p**

```
Point2 p = new Point2();
p.set(-1.0, 1.5);
```

might be something like (-1.0, 1.5)

```
/**  
 * Returns a string representation of this point. The string  
 * representation of this point is the x and y-coordinates  
 * of this point, separated by a comma and space, inside a pair  
 * of parentheses.  
 *  
 * @return a string representation of this point  
 */  
  
@Override  
public String toString() {  
    return "(" + this.x + ", " + this.y + ")";  
}
```

@Override is an optional annotation that we can use to tell the compiler that we are redefining the behavior of the **toString** method that was inherited from **java.lang.Object**

toString

- ▶ by providing **toString** clients can now easily get a string representation of a **Point2** object

```
import ca.queensu.cs.cisc124.notes.basics.geometry.Point2;

public class PrintPoint {

    public static void main(String[] args) {
        // create a point
        Point2 p = new Point2();

        // set its coordinates
        p.set(-1.0, 1.5);

        // print the point
        System.out.println(p.toString());
    }
}
```

Counter methods

- ▶ for our **Counter** class we would like to provide methods that:
 - ▶ get the current value of the counter
 - ▶ advances the value of the counter by 1
 - ▶ returns a string representation of the counter

```
package ca.queensu.cs.cisc124.notes.basics.counter;

public class Counter {

    private int value;

    /**
     * Initializes this counter so that its current value is 0.
     */
    public Counter() {
        this.value = 0;
    }

    /**
     * Returns the current value of this counter.
     *
     * @return the current value of this counter
     */
    public int value() {
        return this.value;
    }
}
```

```
/**  
 * Increment the value of this counter upwards by 1. If this  
 * method is called when the current value of this counter is  
 * equal to {@code Integer.MAX_VALUE} then the value of this  
 * counter is set to 0 (i.e., the counter wraps around to 0).  
 */  
public void advance() {  
    if (this.value != Integer.MAX_VALUE) {  
        this.value++;  
    }  
    else {  
        this.value = 0;  
    }  
}
```

```
/*
 * Returns a string representation of this counter. The string
 * representation is the string {@code "count: "} followed
 * by the current value of this counter.
 *
 * @return a string representation of this counter
 */
@Override
public String toString() {
    return "count: " + this.value;
}

} // end of Counter class
```

Stopwatch methods

- ▶ for our **Stopwatch** class we would like to provide methods that:
 - ▶ starts the watch
 - ▶ stops the watch
 - ▶ gets the elapsed time in seconds
 - ▶ if the watch is running, then the elapsed time is the time from when the watch was started to now
 - ▶ if the watch is not running, then the elapsed time is the time from when the watch was started to when it was stopped

```
package ca.queensu.cs.cisc124.notes.basics.stopwatch;

public class Stopwatch {

    private long rawStartTime;
    private long rawStopTime;
    private boolean isRunning;

    /**
     * Number of nanoseconds in one second
     */
    private final static long BILLION = 1000000000L;

    /**
     * Initializes this stopwatch so that the elapsed time
     * is zero seconds.
     */
    public Stopwatch() {
        this.rawStartTime = 0L;
        this.rawStopTime = 0L;
        this.isRunning = false;
    }
}
```

```
/**  
 * Starts this stopwatch running. Does nothing if this  
 * stopwatch is already running.  
 *  
 */  
public void start() {  
    if (!this.isRunning) {  
        this.rawStartTime = System.nanoTime();  
        this.isRunning = true;  
    }  
}
```

```
/**  
 * Stops this stopwatch and returns the elapsed time in  
 * seconds since the watch was started. If the stopwatch is  
 * already stopped then the elapsed time in seconds (as  
 * computed by {@code elapsed}) is returned.  
 *  
 * @return the elapsed time in seconds  
 */  
  
public double stop() {  
    if (this.isRunning) {  
        this.rawStopTime = System.nanoTime();  
        this.isRunning = false;  
    }  
    return this.elapsed();  
}
```

```
/**  
 * For a running stopwatch, this method returns the total  
 * amount of time in seconds that have elapsed since the  
 * stopwatch was started.  
 *  
 * <p>  
 * For a stopped stopwatch, this method returns the total  
 * amount of time in seconds that elapsed between starting and  
 * stopping the stopwatch.  
 *  
 * @return the elapsed time recorded by this stopwatch  
 */  
  
public double elapsed() {  
    long currentTime = System.nanoTime();  
    if (!this.isRunning) {  
        currentTime = this.rawStopTime;  
    }  
    return (0.0 + currentTime - this.rawStartTime) / BILLION;  
}
```

Constructors in greater detail

Custom constructors

- ▶ a class can have multiple constructors but the signatures of the constructors must be unique
 - ▶ i.e., each constructor must have a unique list of parameter types
- ▶ it would be convenient for clients if **Point2** had a constructor that let the client set the x and y coordinate of the point

```
package ca.queensu.cs.cisc124.notes.basics.geometry;

public class Point2 {
    private double x;
    private double y;

    // no-argument constructor not shown

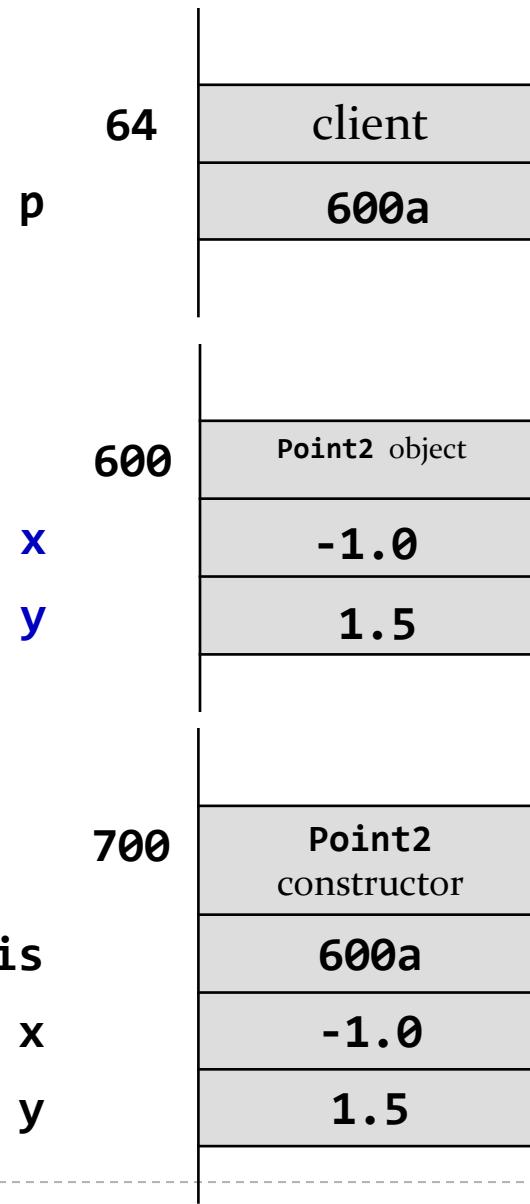
    /**
     * Initializes the elements of this point to {@code (x, y)} where
     * {@code x} and {@code y} are specified by the caller.
     *
     * @param x the x value of this point
     * @param y the y value of this point
     */
    public Point2(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

this.x : the field named **x** of **this** point
this.y : the field named **y** of **this** point
x : the parameter named **x** of the constructor
y : the parameter named **y** of the constructor

```
Point2 p = new Point2(-1.0, 1.5);
```

1. a reference variable **p** is created somewhere in memory
2. **new** allocates memory for a **Point2** object
3. the **Point2** constructor is invoked by passing the memory address of the object and the arguments **-1.0** and **1.5** to the constructor
4. the constructor runs, setting the values of the fields **this.x** and **this.y**
5. the value of **p** is set to the memory address of the constructed object

fields {



this
parameters {
 x
 y

this

- ▶ in our constructor

```
public Point2(double x, double y) {  
    this.x = x;  
    this.y = y;  
}
```

there are parameters with the same names as fields

- ▶ when this occurs, the parameter has precedence over the field
 - ▶ we say that the parameter *shadows* the field
 - ▶ when shadowing occurs you must use **this** to refer to the field

Custom constructors

- ▶ adding the constructor **Point2(double x, double y)** makes it easier for users to create points

```
package lectures.basics;

import lectures.basics.geometry.Point2;

public class PrintPoint {

    public static void main(String[] args) {
        // create a point
        Point2 p = new Point2();

        // set its coordinates
        p.set(-1.0, 1.5);

        // print the point
        System.out.println(p.toString());
    }
}
```

```
package lectures.ctor;

import lectures.ctor.geometry.Point2;

public class PrintPoint {

    public static void main(String[] args) {
        // create a point
        Point2 p = new Point2(-1.0, 1.5);

        // print the point
        System.out.println(p.toString());
    }
}
```

Copy constructor

- ▶ a copy constructor initializes the state of an object by copying the state of another object (having the same type)
- ▶ it has a single parameter that is the same type as the class

```
/**  
 * Initializes the coordinates of this point by copying the  
 * coordinates from {@code other}.  
 *  
 * @param other the point to copy  
 */  
public Point2(Point2 other) {  
    this.x = other.x;  
    this.y = other.y;  
}
```

Copy constructor

- ▶ adding a copy constructor allows the client to simplify their code

```
import ca.queensu.cs.cisc124.notes.basics.geometry.Point2;

public class CopyPoint {

    public static void main(String[] args) {
        // create a point and print it
        Point2 p = new Point2(-1.0, 1.5);
        System.out.println("p: " + p.toString());

        // copy p and print q
        Point2 q = new Point2(p);
        System.out.println("q: " + q.toString());

        // mutate p
        p.set(99.0, 44.0);

        // print p and q
        System.out.println("p: " + p.toString());
        System.out.println("q: " + q.toString());
    }
}

p: (-1.0, 1.5)
q: (-1.0, 1.5)
p: (99.0, 44.0)
q: (-1.0, 1.5)
```

Avoiding Code Duplication

- ▶ notice that the constructor bodies are almost identical to each other
 - ▶ all three constructors have 2 lines of code
 - ▶ all three constructors set the x and y coordinate of the point
- ▶ whenever you see duplicated code you should consider moving the duplicated code into a method
- ▶ in this case, one of the constructors already does everything we need to implement the other constructors...

Constructor chaining

- ▶ a constructor is allowed to invoke another constructor
- ▶ when a constructor invokes another constructor it is called *constructor chaining*
- ▶ to invoke a constructor in the same class you use the **this** keyword
 - ▶ if you do this then it *must occur* on the first line of the constructor body
 - ▶ but you *cannot* use **this** in a method to invoke a constructor
- ▶ we can re-write two of our constructors to use constructor chaining...

```
public class Point2 {  
    private double x;  
    private double y;
```

```
    public Point2() {  
        this(0.0, 0.0);  
    }
```

invokes

```
    public Point2(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }
```



```
    public Point2(Point2 other) {  
        this(other.x, other.y);  
    }
```

invokes



Counter constructors

- ▶ we can add additional constructors to the **Counter** class:
 - ▶ a constructor that initializes the counter to some specified positive value
 - ▶ a copy constructor
- ▶ we can also use constructor chaining to minimize code duplication

```
package ca.queensu.cs.cisc124.notes.basics.counter;

public class Counter {

    private int value;

    /**
     * Initializes this counter so that its current value is 0.
     */
    public Counter() {
        this(0); // ctor chaining to minimize code duplication
    }
}
```

```
/**  
 * Initializes this counter to the specified non-negative  
 * value.  
 *  
 * @param value the starting value of this counter  
 * @throws IllegalArgumentException if value is negative  
 */  
public Counter(int value) {  
    if (value < 0) {  
        throw new  
            IllegalArgumentException("value must be non-negative");  
    }  
    this.value = value;  
}
```

```
/**  
 * Initializes this counter so that its current value is equal  
 * to the current value of {@code other}.  
 *  
 * @param other the counter to copy the value from  
 */  
public Counter(Counter other) {  
    this(other.value); // ctor chaining to minimize code  
                      // duplication  
}
```

Classes continued

Documenting

- ▶ documenting code was not a new idea when Java was invented
- ▶ however, Java was the first major language to embed documentation in the code and extract the documentation into readable electronic APIs
- ▶ the tool that generates API documents from comments embedded in the code is called Javadoc

Javadoc

- ▶ Javadoc processes *doc comments* that immediately precede a class, attribute, constructor or method declaration
 - ▶ doc comments delimited by `/**` and `*/`
 - ▶ doc comment written in HTML and made up of two parts
 1. a description
 - first sentence of description gets copied to the summary section
 - only one description block; can use `<p>` to create separate paragraphs
 2. block tags
 - begin with `@` (`@param`, `@return`, `@throws` and many others)

Javadoc

- ▶ every class should have a Javadoc comment that describes the purpose of the class

```
package ca.queensu.cs.cisc124.notes.basics.geometry;

/**
 * A Cartesian point in 2-dimensions having real
 * coordinates.
 */
public class Point2 {
    /**
     * The coordinates of this point.
     */
    private double x;
    private double y;
}
```

Package `lectures.basics.geometry`

Class Point2

`java.lang.Object`
`lectures.basics.geometry.Point2`

```
public class Point2
extends Object
```

A Cartesian point in 2-dimensions having real coordinates.

Constructor Summary

Constructors

Constructor	Description
-------------	-------------

Javadoc

- ▶ every public field should have a Javadoc comment that describes the purpose of the field
- ▶ the first sentence of the Javadoc comment will appear in the *Field Summary* section of the generated documentation

```
// Example from java.lang.Math

public class Math {

    /**
     * The double value that is closer than any other to e, the
     * base of the natural logarithms.
     */
    public static final double E = 2.718281828459045;

    /**
     * The double value that is closer than any other to pi,
     * the ratio of the circumference of a circle to its diameter.
     */
    public static final double PI = 3.141592653589793;
```

Field Summary

Fields

Modifier and Type	Field and Description
static double	E The double value that is closer than any other to <i>e</i> , the base of the natural logarithms.
static double	PI The double value that is closer than any other to <i>pi</i> , the ratio of the circumference of a circle to its diameter.

Method Summary

All Methods

Static Methods

Concrete Methods

Modifier and Type

Method and Description

Javadoc

- ▶ every public constructor should have a Javadoc comment that describes the purpose of the constructor
- ▶ the first sentence of the Javadoc comment will appear in the *Constructor Summary* section of the generated documentation
- ▶ the complete documentation appears in the *Constructor Detail* section of the generated documentation

```
/**  
 * Initializes the coordinates of this point to  
 * {@code (0.0, 0.0)}.  
 */  
public Point2() {  
    this.x = 0.0;  
    this.y = 0.0;  
}
```

Javadoc

- ▶ if a constructor has parameters then each parameter should be described using an `@param` tag

```
/**  
 * Initializes the elements of this point to {@code (x, y)} where  
 * {@code x} and {@code y} are specified by the caller.  
 *  
 * @param x the x value of this point  
 * @param y the y value of this point  
 */  
public Point2(double x, double y) {  
    this.x = x;  
    this.y = y;  
}
```

```
/**  
 * Initializes the coordinates of this point by copying the  
 * coordinates from {@code other}.  
 *  
 * @param other the point to copy  
 */  
public Point2(Point2 other) {  
    this.x = other.x;  
    this.y = other.y;  
}
```

Constructor Summary

Constructors

Constructor	Description
<code>Point2()</code>	Initializes the coordinates of this point to (0.0, 0.0).
<code>Point2(double x, double y)</code>	Initializes the coordinates of this point to (x, y) where x and y are specified by the caller.
<code>Point2(Point2 other)</code>	Initializes the coordinates of this point by copying the coordinates from other.

Method Summary



Constructor Detail

Point2

```
public Point2()
```

Initializes the coordinates of this point to (0.0, 0.0).

Point2

```
public Point2(double x,  
             double y)
```

Initializes the coordinates of this point to (x, y) where x and y are specified by the caller.

Parameters:

x - the x value of this point

y - the y value of this point

Point2

```
public Point2(Point2 other)
```

Initializes the coordinates of this point by copying the coordinates from `other`.

Parameters:

`other` - the point to copy

Javadoc

- ▶ every public method should have a Javadoc comment that describes the purpose of the method
- ▶ the first sentence of the Javadoc comment will appear in the *Method Summary* section of the generated documentation
- ▶ the complete documentation appears in the *Method Detail* section of the generated documentation

Javadoc

- ▶ if a method has parameters then each parameter should be described using an `@param` tag
- ▶ if a method returns a value then the return value should be described using an `@return` tag

```
/**  
 * Sets the x and y coordinate to the specified values.  
 *  
 * @param newX the new x coordinate  
 * @param newY the new y coordinate  
 * @return a reference to this point  
 */  
public Point2 set(double newX, double newY) {  
    this.x = newX;  
    this.y = newY;  
    return this;  
}
```

set

```
public Point2 set(double newX, double newY)
```

Sets the x and y coordinate to the specified values.

Parameters:

newX - the new x coordinate

newY - the new y coordinate

Returns:

a reference to this coordinate

Javadoc

- ▶ if a constructor or method throws an exception then the exception and the conditions under which it is thrown should be described using a **@throws** tag

```
/**  
 * Initializes this counter to the specified non-negative  
 * value.  
 *  
 * @param value the starting value of this counter  
 * @throws IllegalArgumentException if value is negative  
 */  
public Counter(int value) {  
    if (value < 0) {  
        throw new IllegalArgumentException("value must be non-  
negative");  
    }  
    this.value = value;  
}
```

Counter

```
public Counter(int value)
```

Initializes this counter to the specified non-negative value.

Parameters:

value - the starting value of this counter

Throws:

IllegalArgumentException - if value is negative

Javadoc

- ▶ you may use a subset of HTML tags in Javadoc comments
- ▶ it is common to use the paragraph tag <p> to add paragraphs to the generated documentation

```
/**  
 * Add a vector to this point changing the coordinates of  
 * this point.  
 *  
 * <p>  
 * Use this method when you want to write something like  
 * {@code p = p + v} where {@code p} is a point and {@code v}  
 * is a vector.  
 *  
 * @param v the vector to add  
 * @return a reference to this point  
 */  
public Point2 add(Vector2 v) {
```

add

```
public Point2 add(Vector2 v)
```

Add a vector to this point changing the coordinates of this point.

Use this method when you want to write something like $p = p + v$ where p is a point and v is a vector.

Parameters:

v - the vector to add

Returns:

a reference to this point

Javadoc

- ▶ if you plan on writing Java code professionally you should familiarize yourself with the recommendations for writing Javadoc comments:
- ▶ <https://www.oracle.com/technetwork/java/javase/tech/index-137868.html>

Preconditions & Postconditions

Method preconditions

- ▶ some methods require that one or more conditions be true before the method is run
 - ▶ e.g., **divide** requires that **s** not be equal to zero
- ▶ method precondition
 - ▶ a condition that the *caller of the method* must ensure is true immediately before calling the method
 - ▶ note that the responsibility for meeting a precondition lies with the caller of the method

Method preconditions

- ▶ if a method has a precondition, then the precondition should be documented in the Javadoc for the method
- ▶ unfortunately, Javadoc has no dedicated tag for preconditions
 - ▶ the precondition must be documented in the method description or in the parameter description

```
/**  
 * Divide this point by a scalar value changing the coordinates  
 * of this point.  
 *  
 * <p>  
 * Use this method when you want to write something like  
 * {@code p = p / s} where {@code p} is a point and {@code s}  
 * is a scalar.  
 *  
 * @param s the scalar value to divide this point by; must  
 * not be equal to zero  
 * @return a reference to this point  
 */  
public Point2 divide(double s) {
```

Method preconditions

- ▶ because the responsibility for meeting a precondition lies with the caller of the method the implementer of the method is allowed to do anything when a precondition is not satisfied
- ▶ for example, the implementer of divide can ignore the case where **s == 0** is true

```
/**  
 * Divide this point by a scalar value changing the coordinates  
 * of this point.  
 *  
 * <p>  
 * Use this method when you want to write something like  
 * {@code p = p / s} where {@code p} is a point and {@code s}  
 * is a scalar.  
 *  
 * @param s the scalar value to divide this point by; must  
 * not be equal to zero  
 * @return a reference to this point  
 */  
public void divide(double s) {  
    this.x = this.x / s;  
    this.y = this.y / s;  
}
```

Method preconditions

- ▶ the problem with assuming that the precondition is true is that the method will run with an input argument that is invalid for the method
- ▶ possible consequences
 - ▶ method returns the wrong value
 - ▶ method encounters an error (possibly throwing an unexpected exception)
 - ▶ method does not complete everything in its contract
 - ▶ it does not satisfy its postconditions
 - ▶ method puts the object into an invalid state

```
public static void main(String[] args) {  
    Point2 p = new Point2(1.0, 2.0);  
    p.divide(0.0);  
    System.out.println(p);  
}
```

prints

(Infinity, Infinity)

Method preconditions

- ▶ preconditions on **public** methods should be enforced by explicitly checking the argument values
- ▶ a method should throw an exception to indicate that precondition has not been satisfied

Exceptions

- ▶ the term exception is shorthand for “exceptional event”
- ▶ “When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called *throwing an exception*.”

The Java Tutorials: Exceptions

<https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html>

Exceptions

- ▶ to throw an exception the programmer makes an exception object
- ▶ common classes that define exceptions:
 - ▶ **IllegalArgumentException**
 - ▶ **IndexOutOfBoundsException**
 - ▶ **NullPointerException**

Exceptions

- ▶ after making the exception object, use the keyword **throws** followed by a reference to the exception object to throw the exception
- ▶ throwing an exception causes the method to immediately stop running after the **throws** statement
 - ▶ i.e., any code in the method after the **throws** statement never runs
 - ▶ this means that you should always check the arguments first and throw an exception if required *before* modifying the state of the object

```
/**  
 * Divide this point by a scalar value changing the coordinates  
 * of this point.  
 * ...  
 *  
 * @param s the scalar value to divide this point by  
 * @return a reference to this point  
 * @throws IllegalArgumentException if s == 0.0 is true  
 */  
public Point2 divide(double s) {  
    if (s == 0.0) {  
        throw new IllegalArgumentException("division by 0.0");  
    }  
    this.x /= s;  
    this.y /= s;  
    return this;  
}
```

Postconditions

- ▶ method postcondition
 - ▶ a condition that the method must ensure is true immediately after the method finishes running
 - ▶ note that the responsibility for meeting a postcondition lies with the method
 - ▶ if the postcondition is not true, then there is something wrong with the implementation of the method

Postconditions

- ▶ for a non-static method postconditions can involve:
 - ▶ the value returned by the method (if any)
 - ▶ the fields of the object (for mutator methods)
 - ▶ the arguments passed to the method

- ▶ in Java postconditions are normally described in the documentation of the method

```
/**  
 * Returns the x coordinate of this point.  
 *  
 * @return the x coordinate of this point  
 */  
public double x() {  
    return this.x;  
}
```

Postconditions

- ▶ it is possible but usually awkward to express a postcondition as an actual condition in Java
- ▶ the complete postcondition for the previous method is surprisingly complicated

```
/**  
 * Returns the x coordinate of this point.  
 *  
 * @return the x coordinate of this point  
 */  
  
public double x() {  
    final double oldX = this.x;  
    final double oldY = this.y;  
  
    final double result = this.x;  
    return result;  
  
    // postcondition: this.x == oldX && this.y == old.y &&  
    //                  this.x == result  
}
```

In Java, the keyword **final** means that the variable may be assigned a value only once in its lifetime.

Postconditions

- ▶ the postconditions for a mutator method involve the fields of the object

```
/**  
 * Sets the x coordinate to the specified value.  
 *  
 * @param newX the new x coordinate  
 * @return a reference to this point  
 */  
public Point2 x(double newX) {  
    this.x = newX;  
    return this;  
}
```

Postconditions

- ▶ in the previous example, the postcondition can be expressed as a condition
- ▶ but is surprisingly complicated for such a simple method

```
/**  
 * Sets the x coordinate to the specified value.  
 *  
 * @param newX  
 *         the new x coordinate of the point  
 */  
public Point2 x(double newX) {  
    final double paramX = newX;  
    final double oldY = this.y;  
  
    this.x = newX;  
    final Point2 result = this;  
    return result;  
  
    // postcondition: this.x == paramX && this.y == oldY &&  
    //                      this == result  
}
```

```
/**  
 * Sets the x coordinate to the specified value.  
 *  
 * @param newX  
 *         the new x coordinate of the point  
 */  
public Point2 x(double newX) {  
    final double paramX = newX;  
    final double oldY = this.y;  
  
    newX = 1;  
    this.x = newX;  
    final Point2 result = this;  
    return result;  
  
    // postcondition: this.x == paramX && this.y == oldY &&  
    //                      this == result  
}
```

Postconditions

- ▶ is there any way to check the postcondition?
- ▶ you could use an if statement to check the postcondition and throw an exception if the postcondition is false
 - ▶ this imposes a cost on every caller of the method
 - ▶ i.e., the caller of the method must wait for the postcondition to be checked

```
public Point2 set(double newX, double newY) {  
    final double paramX = newX;  
    final double oldY = this.y;  
  
    newX = 1;  
    this.x = newX;  
    final Point2 result = this;  
  
    if (!(this.x == paramX && this.y == oldY &&  
        this == result)) {  
        throw new RuntimeException("postcondition is false");  
    }  
  
    return result;  
}
```

Postconditions

- ▶ an alternative is to use an **assert** statement
- ▶ an **assert** statement has a condition that you think is true
 - ▶ if the condition is not true then an error is thrown
- ▶ assertions can be turned on or off when the program is compiled
 - ▶ turn the assertions on when you are developing and testing your program
 - ▶ turn the assertions off when you are confident your program is error free

```
public Point2 set(double newX, double newY) {  
    final double paramX = newX;  
    final double oldY = this.y;  
  
    newX = 1;  
    this.x = newX;  
    final Point2 result = this;  
  
    assert (this.x == paramX && this.y == oldY &&  
           this == result);  
  
    return result;  
}
```

Postconditions

- ▶ assertions are turned off by default in eclipse
- ▶ you can enable assertions by editing the run configuration for your program
 1. Run » Run Configurations...
 2. choose the Java Application that you want to turn assertions on for in the left-hand panel
 3. click on the Arguments tab in the right-hand panel
 4. add the flag **-ea** in the VM arguments
 5. click Apply
 6. click Run to run the application or click Close to close the Run Configurations dialog

Class Invariants

Class invariants

- ▶ an *invariant* is a condition that is always true
- ▶ a *class invariant* is a condition regarding the state of a
an object that is always true
- ▶ the invariant must be established when the object is created
and every public method of the class must ensure that the
invariant is true when the method finishes running

Class invariants

- ▶ some classes have no “interesting” invariants
 - ▶ e.g., **Point2** has no interesting invariants

Class invariant of Counter

- ▶ **Counter** has an important class invariant:

`c.value() >= 0 &&`

`c.value() <= Integer.MAX_VALUE`

- ▶ you should examine the implementation of **Counter** to check that all of the constructors and public methods ensure that the invariant is true

Checking class invariants

- ▶ class invariants must be true at the end of every constructor and public method
- ▶ it is possible to check that the invariants are true using **assert**

```
package lectures.invariants.counter;

/**
 * The {@code Counter} class represents a device used to
 * incrementally count upwards from zero up to
 * {@link java.lang.Integer#MAX_VALUE}.
 *
 */
public class Counter {

    /**
     * The current value of this counter.
     */
    private int value;

    /**
     * Asserts that the class invariant is true.
     */
    private void checkInvariant() {
        assert this.value() >= 0;
    }
}
```

```
/**  
 * Initializes this counter so that its current value is 0.  
 */  
public Counter() {  
    this(0); // constructor chaining to minimize code duplication  
  
    this.checkInvariant();  
}
```

```
/**  
 * Initializes this counter to the specified non-negative  
 * value.  
 *  
 * @param value the starting value of this counter  
 * @throws IllegalArgumentException if value is negative  
 */  
public Counter(int value) {  
    if (value < 0) {  
        throw new IllegalArgumentException("value must be non-  
negative");  
    }  
    this.value = value;  
  
    this.checkInvariant();  
}
```

```
/**  
 * Initializes this counter so that its current value is equal  
 * to the current value of {@code other}.  
 *  
 * @param other the counter to copy the value from  
 */  
public Counter(Counter other) {  
    this(other.value);  
  
    this.checkInvariant();  
}
```

```
/**  
 * Returns the current value of this counter.  
 *  
 * @return the current value of this counter  
 */  
public int value() {  
    this.checkInvariant(); // not allowed after return statement  
    return this.value;  
}
```

```
/**  
 * Increment the value of this counter upwards by 1. If this  
 * method is called when the current value of this counter is  
 * equal to {@code Integer.MAX_VALUE} then the value of this  
 * counter is set to 0 (i.e., the counter wraps around to  
 * 0).  
 */  
public void advance() {  
    if (this.value != Integer.MAX_VALUE) {  
        this.value++;  
    } else {  
        this.value = 0;  
    }  
    this.checkInvariant();  
}
```

```
/**  
 * Returns a string representation of this counter. The string  
 * representation is the string {@code "count: "} followed by  
 * the current value of this counter.  
 *  
 * @return a string representation of this counter  
 */  
  
@Override  
public String toString() {  
    this.checkInvariant(); // not allowed after return statement  
    return "count: " + this.value;  
}
```

Static features

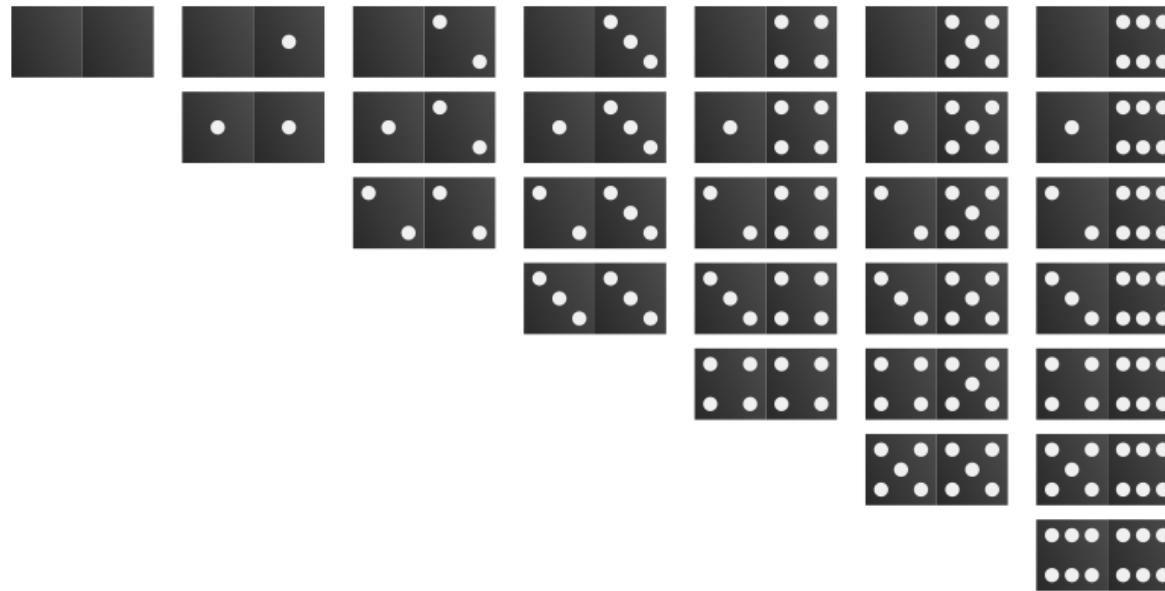
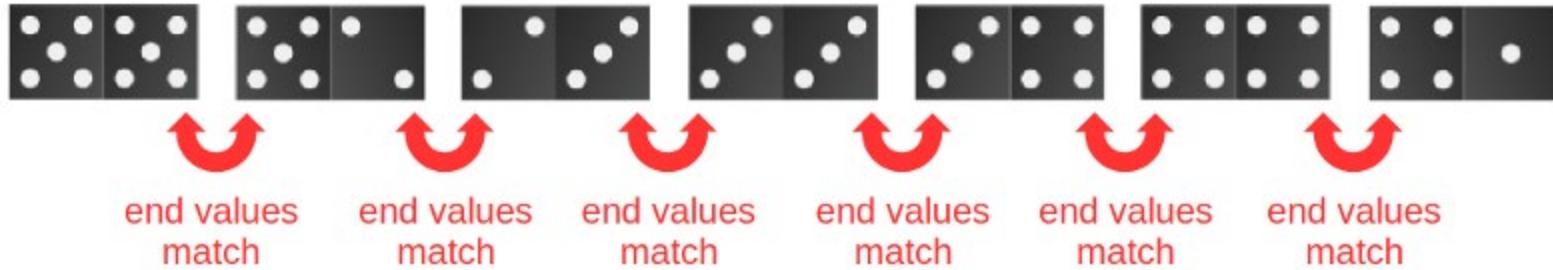
Static features

- ▶ a class can have fields and methods that are **static**
- ▶ static features are associated with the class instead of individual objects

static Fields

- ▶ a field that is **static** is a per-class member
 - ▶ only one copy of the field, and the field is associated with the class
 - ▶ every object created from a class declaring a static field shares the same copy of the field
 - ▶ also commonly called *class variable*

Dominoes



```
public class Domino {  
  
    /**  
     * The smallest possible value for a side of a domino.  
     */  
    public static final int MIN_VALUE = 0;  
  
    /**  
     * The largest possible value for a side of a domino.  
     */  
    public static final int MAX_VALUE = 6;  
  
    private int val1;  
    private int val2;
```

static Fields

```
Domino d1 = new Domino(1, 3);  
Domino d2 = new Domino(5, 5);
```

d1

d2

MIN_VALUE

MAX_VALUE

val1

val2

val1

val2

64

500

1000

1100

client invocation

1000a

1100a

Domino class

0

6

Domino object

1

3

Domino object

5

5

static Field Client Access

- ▶ a client should access a **public static** field without using an object reference
- ▶ use the class name followed by a period followed by the field name

```
public static void main(String[] args) {  
  
    System.out.println("minimum domino value " + Domino.MIN_VALUE);  
}
```

static Field Client Access

- ▶ it is legal, *but considered bad form*, to access a **public static** field using an object reference

```
public static void main(String[] args) {  
  
    Domino d = new Domino(3, 4);  
    System.out.println("minimum domino value " + d.MIN_VALUE);  
}
```

final Fields

- ▶ a field that is **final** can only be assigned to once
- ▶ **public static final** fields are typically assigned when they are declared

```
public static final int MIN_VALUE = 0;
```

- ▶ **public static final** fields are intended to be constant values that are a meaningful part of the abstraction provided by the class

final Fields of Primitive Types

- ▶ **final** fields of primitive types are constant

```
public class Domino {  
    public static final int MIN_VALUE = 0;  
}
```

```
// client of Domino  
public static void main(String[] args) {  
  
    Domino.MIN_VALUE = 100; // will not compile;  
                          // field MIN_VALUE  
                          // is final and  
                          // previously assigned  
}
```

final Fields of Immutable Types

- ▶ **final** fields of immutable types are constant

```
public class NothingToHide {  
    public static final String X = "peek-a-boo";  
}
```

```
// client of NothingToHide  
public static void main(String[] args) {  
    NothingToHide.X = "i-see-you";  
        // will not compile;  
        // field X is final and  
        // previously assigned  
}
```

- ▶ **String** is immutable
 - ▶ it has no methods to change its contents

final Fields of Mutable Types

- ▶ **final** fields of mutable types are not logically constant; their state can be changed

```
public class ReallyNothingToHide {  
    public static final List<String> EMPTY =  
        new ArrayList<>();  
}
```

```
// client of ReallyNothingToHide  
public static void main(String[] args) {  
    ReallyNothingToHide.EMPTY.add("oops");  
        // works!!  
        // EMPTY is now not empty  
}
```

final fields

- ▶ avoid using mutable types as **public** constants
 - ▶ they are not logically constant

static methods

- ▶ a method that is **static** is a per-class member
 - ▶ client does not need an object reference to invoke the method
 - ▶ client uses the class name to access the method
 - ▶ **static** methods are also called *class methods*

```
/**  
 * Returns true if the specified value is a legal domino value,  
 * and false otherwise.  
 *  
 * @param value  
 *         a value to check  
 * @return true if the specified value is a legal domino value,  
 *         and false otherwise  
 */  
public static boolean isValueOK(int value) {  
    return value >= Domino.MIN_VALUE &&  
        value <= Domino.MAX_VALUE;  
}
```

static Method Client Access

- ▶ a client should access a **public static** method without using an object reference
- ▶ use the class name followed by a period followed by the field name

```
public static void main(String[] args) {  
  
    System.out.println(Domino.isValueOK(8));  
}
```

static Method Client Access

- ▶ it is legal, *but considered bad form*, to access a **public static** method using an object reference

```
public static void main(String[] args) {  
  
    Domino d = new Domino(3, 4);  
    System.out.println(d.isValueOK(8));  
}
```

Utility classes

Review: Java Class

- ▶ a class is a model of a thing or concept
- ▶ in Java, a class is usually a blueprint for creating objects
 - ▶ fields (or attributes)
 - ▶ the structure of an object; its components and the information (data) contained by the object
 - ▶ methods
 - ▶ the behaviour of an object; what an object can do

Utility classes

- ▶ sometimes, it is useful to create a class called a *utility class* that is not used to create objects
 - ▶ such classes have no constructors for a client to use to create objects
- ▶ in a utility class, all features are marked as being **static**
 - ▶ you use the class name to access these features
- ▶ examples of utility classes:
 - ▶ **java.lang.Math**
 - ▶ **java.util.Arrays**
 - ▶ **java.util.Collections**

Utility classes

- ▶ the purpose of a utility class is to group together related fields and methods where creating an object is not necessary
- ▶ **java.lang.Math**
 - ▶ groups mathematical constants and functions
 - ▶ do not need a **Math** object to compute the cosine of a number
- ▶ **java.util.Collections**
 - ▶ groups methods that operate on Java collections
 - ▶ do not need a **Collections** object to sort an existing **List**

Class versus utility class

- ▶ a utility class is *never* used to create objects
- ▶ when you use a utility class only the class itself occupies any memory

```
public static void main(String[] args) {  
  
    double x = Math.cos(Math.PI / 3.0);  
    double y = Math.sin(Math.PI / 3.0);  
  
    // notice that we never created a Math object  
}
```

Name	Address
	100 Math class
PI	3.1415....
E	2.7182....
x	200 main method
	0.8660....
y	0.5

Math class is loaded into memory but there are no **Math** instances

the value **cos(π/3)**

the value **sin(π/3)**

these are values (not addresses) because **x** and **y** are primitive variables (**double**)

A simple utility class

- ▶ implement a utility class that helps you calculate Einstein's famous mass-energy equivalence equation $E = mc^2$ where
 - ▶ m is mass (in kilograms)
 - ▶ c is the speed of light (in metres per second)
 - ▶ E is energy (in joules)

Start by creating a package, giving the class a name, and creating the class body block.

```
public class Relativity {
```

```
}
```

Add a field that represents the speed of light.

```
public class Relativity {  
    public static final double C = 299792458;  
}
```

Add a method to compute $E = mc^2$.

```
public class Relativity {  
  
    public static final double C = 299792458;  
  
    public static double massEnergy(double mass) {  
        double energy = mass * Relativity.C * Relativity.C;  
        return energy;  
    }  
  
}
```

Here's a program that uses (a client) the **Relativity** utility class.

```
public class OneGram {  
  
    public static void main(String[] args) {  
        double mass = 0.001;  
        double energy = Relativity.massEnergy(mass);  
        System.out.println("1 gram = " + energy + " Joules");  
    }  
  
}  
  
1 gram = 8.987551787368177E13 Joules
```

new Relativity objects

- ▶ our **Relativity** class does not expose a constructor
 - ▶ but

```
Relativity y = new Relativity();
```

is legal

- ▶ if you do not define any constructors, Java will generate a default no-argument constructor for you
 - ▶ e.g., we get the **public** constructor

```
public Relativity() { }
```

even though we did not implement it

Preventing instantiation

- ▶ in a utility class you can prevent a client from making new instances of your class by declaring a **private** constructor
- ▶ a **private** field, constructor, or method can only be used inside the class that it is declared in

```
public class Relativity {  
  
    public static final double C = 299792458;  
  
    private Relativity() {  
        // private and empty by design  
    }  
  
    public static double massEnergy(double mass) {  
        double energy = mass * Relativity.C * Relativity.C;  
        return energy;  
    }  
}
```

Preventing instantiation

- ▶ every utility class should have a private empty no-argument constructor to prevent clients from making objects using the utility class

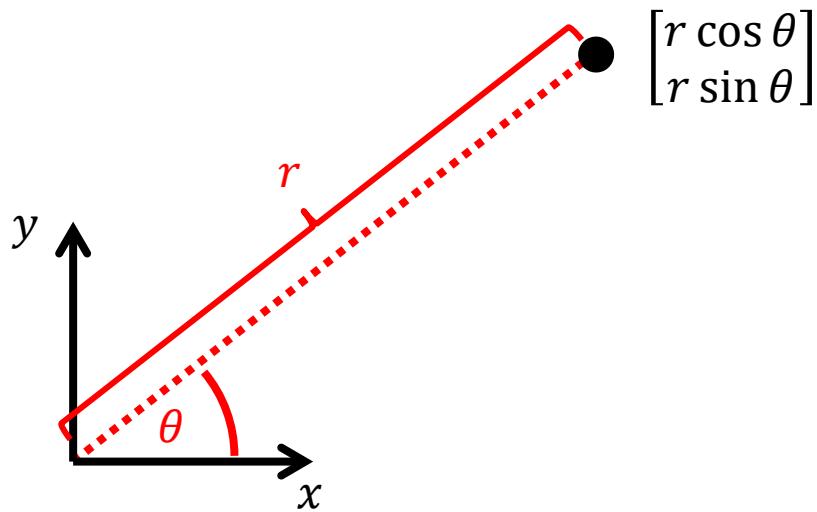
Static factory methods

Static factory methods

- ▶ a common use of static methods in non-utility classes is to create a *static factory method*
- ▶ a static factory method is a static method that returns an instance of the class
- ▶ called a factory method because it makes an object and returns a reference to the object
- ▶ you can use a static factory method to create methods that behave like constructors
 - ▶ they create and return a reference to a new instance
 - ▶ unlike a constructor, the method has a name

Static factory methods

- ▶ recall our point class
- ▶ suppose that you want to provide a constructor that constructs a point given the polar form of the point



```
public class Point2 {
```

```
    private double x;  
    private double y;
```

Illegal overload; both
constructors have the
same signature.

```
    public Point2(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }
```

```
    public Point2(double r, double theta) {  
        this(r * Math.cos(theta), r * Math.sin(theta));  
    }
```

Static factory methods

- ▶ we can eliminate the problem by replacing the second constructor with a static factory method
- ▶ a static factory method is a **static** method that returns a new object
- ▶ also called the *named constructor idiom* because a static factory method can have a descriptive name and it behaves like using **new** to invoke a constructor

```
public class Point2 {  
  
    private double x;  
    private double y;  
  
    public Point2(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

```
public static Point2 polar(double r, double theta) {  
    double x = r * Math.cos(theta);  
    double y = r * Math.sin(theta);  
    return new Point2(x, y);  
}
```

Static factory methods

- ▶ there are many static factory methods in **Point2** and **Vector2**
- ▶ for each of the following non-static methods:
 - ▶ **add**
 - ▶ **subtract**
 - ▶ **multiply**
 - ▶ **divide**
 - ▶ **negate**
- ▶ there is matching static factory method
- ▶ why?

Static factory methods

- ▶ the non-static version of **add** mutates the point used to call the method
- ▶ i.e., the non-static version of add is mathematically equivalent to

$$p = p + v$$

- ▶ the static version of **add** creates a new point without mutating the point (or vector)
- ▶ i.e., the static version of add is mathematically equivalent to

$$q = p + v$$

Static Factory Methods

- ▶ many examples in Java API

- ▶ `java.lang.Integer`

```
public static Integer valueOf(int i)
```

- ▶ Returns a `Integer` instance representing the specified `int` value.

- ▶ `java.util.Arrays`

```
public static int[] copyOf(int[] original, int newLength)
```

- ▶ Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.

- ▶ `java.lang.String`

```
public static String format(String format, Object... args)
```

- ▶ Returns a formatted string using the specified format string and arguments.

equals

equals

- ▶ suppose you write a value class that extends **Object** but you do not override **equals()**
- ▶ what happens when a client tries to use **equals()**?
 - ▶ **Object.equals()** is called

```
// Point2 client

Point2 p = new Point2(1.0, 2.0);
System.out.println( p.equals(p) );           // true

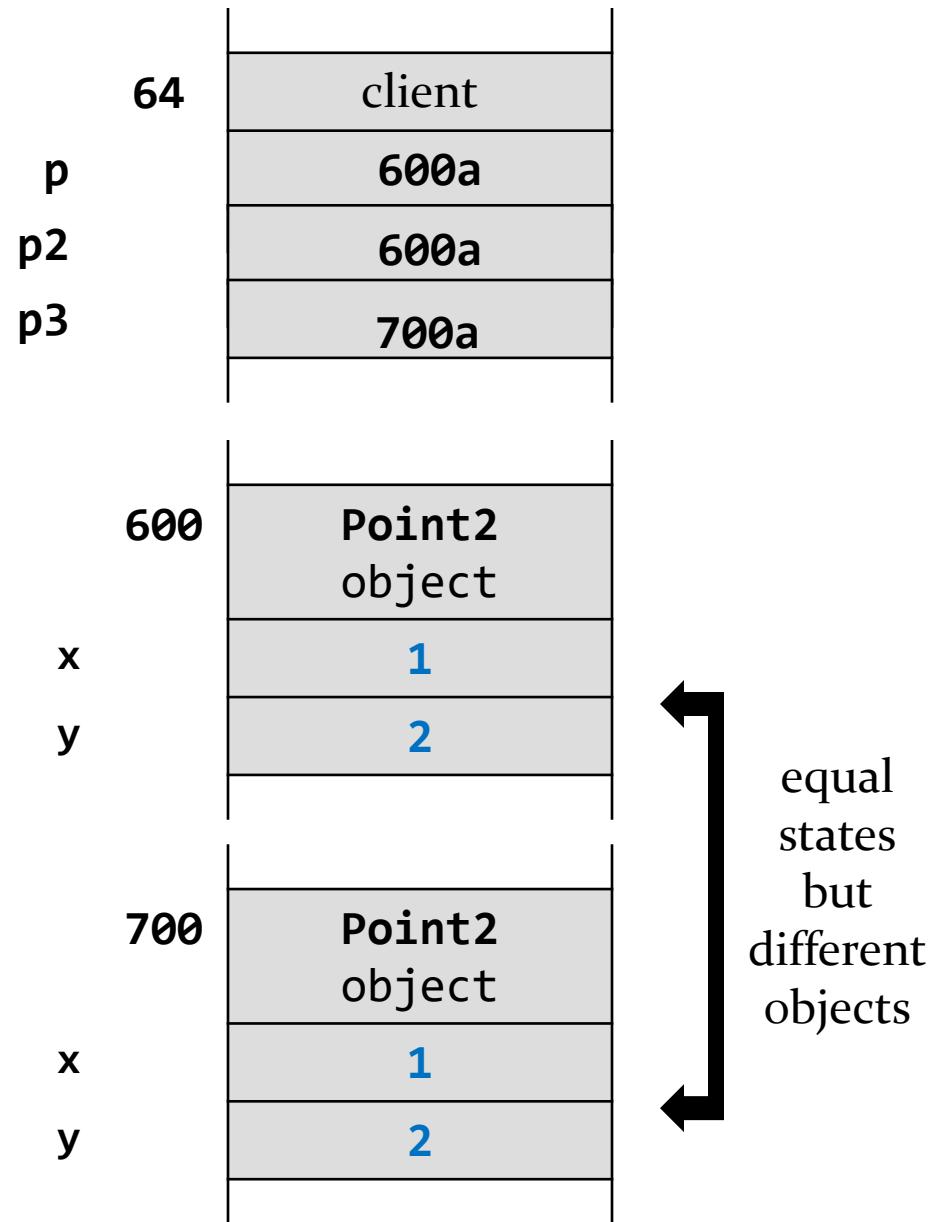
Point2 p2 = p;
System.out.println( p2.equals(p) );           // true

Point2 p3 = new Point2(1.0, 2.0);
System.out.println( p3.equals(p) );           // false!
```

```
Point2 p = new Point2(1.0, 2.0);  
Point2 p2 = p;  
Point2 p3 = new Point2(1.0, 2.0);
```

p and **p2** refer to the object at address 600

p3 refers to the object at address 700



Object.equals

- ▶ **Object.equals** checks if two references refer to the same object
- ▶ **x.equals(y)** is true if and only if **x** and **y** are references to the same object

Point2.equals

- ▶ most value classes should support logical equality
 - ▶ an instance is equal to another instance if their states are equal
 - ▶ e.g. two points are equal if their x and y coordinates both have the same values

- ▶ implementing **equals()** is surprisingly hard
 - ▶ "One would expect that overriding **equals()**, since it is a fairly common task, should be a piece of cake. The reality is far from that. There is an amazing amount of disagreement in the Java community regarding correct implementation of **equals()**. Look into the best Java source code or open an arbitrary Java textbook and take a look at what you find. Chances are good that you will find several different approaches and a variety of recommendations."

□ Angelika Langer, Secrets of equals() – Part 1

- ▶ <http://www.angelikalanger.com/Articles/JavaSolutions/SecretsOfEquals/Equals.html>

- ▶ what we are about to do does not always produce the result you might be looking for
 - ▶ but it is always satisfies the **equals()** contract
 - ▶ and it's what the notes and textbook do

CISC124/CMPE212 Requirements

1. an instance is equal to itself
2. an instance is never equal to **null**
3. today's lecture:
 - ▶ only instances of the exact same type can be equal
4. instances with the same state are equal

1. An Instance is Equal to Itself

- ▶ `x.equals(x)` should always be `true`
- ▶ also, `x.equals(y)` should always be true if `x` and `y` are references to the same object
- ▶ you can check if two references are equal using `==`

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) { — is this object and obj the same object?  
        return true;  
    }  
}
```

Note: This if statement does not need to be here, but the condition is cheap to check

2. An Instance is Never Equal to `null`

- ▶ Java requires that `x.equals(null)` returns `false`
- ▶ and you must not throw an exception if the argument is `null`
- ▶ so it looks like we have to check for a `null` argument...

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj == null) {  
        return false;  
    }  
}
```

Note: The second if statement must be present otherwise a NullPointerException will occur later in the method

3. Instances of the same type can be equal

- ▶ the implementation of **equals()** used in the *Overriding equals* notebook describes a different approach for implementing **equals**
- ▶ we'll revisit this in a later lecture
- ▶ you can find the class of an object using **Object.getClass()**

```
public final Class<? extends Object> getClass()
```

Returns the runtime class of an object.

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj == null) {  
        return false;  
    }  
    if (this.getClass() != obj.getClass()) {  
        return false;  
    }  
}
```

only two objects of
the same type
can be equals

```
}
```

Instances with Same State are Equal

- ▶ recall that the value of the fields of an object define the state of the object
- ▶ two instances are equal if all* of their fields are equal
- ▶ unfortunately, we cannot yet retrieve the attributes of the parameter **obj** because it is declared to be an **Object** in the method signature
- ▶ we need a cast

* not necessarily all of the fields; there are many examples where only some of the fields are used to determine equality

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj == null) {  
        return false;  
    }  
    if (this.getClass() != obj.getClass()) {  
        return false;  
    }  
    Point2 other = (Point2) obj;  
}
```

Cast obj to Simple Point2

Instances with Same State are Equal

- ▶ there is a recipe for checking equality of fields
1. if the field is a primitive type other than **float** or **double** use **==**
 2. if the field type is **float** use
Float.floatToIntBits
 3. if the attribute type is **double** use
Double.doubleToLongBits
 4. if the field is an array consider **Arrays.equals**
 5. if the field is a reference type use **equals**, but beware of fields that might be null
- i.e., safely convert
float to int
double to long

Comparing floating-point values

- ▶ why use `Float.floatToIntBits` and `Double.doubleToLongBits`?
- ▶ because
 - ▶ `Double.NaN == Double.NaN` is **false** whereas

`Double.doubleToLongBits(Double.NaN) ==`
`Double.doubleToLongBits(Double.NaN)` is **true**

- ▶ `-0.0 == 0.0` is **true** whereas

`Double.doubleToLongBits(-0.0) ==`
`Double.doubleToLongBits(0.0)` is **false**

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (this.getClass() != obj.getClass()) {
        return false;
    }
    Point2 other = (Point2) obj;
    if (Double.doubleToLongBits(this.x) != Double.doubleToLongBits(other.x)) {
        return false;
    }
    if (Double.doubleToLongBits(this.y) != Double.doubleToLongBits(other.y)) {
        return false;
    }
    return true;
}
```

equals

- ▶ our version of **equals** compares the state of two points to determine equality
- ▶ now two points with the same coordinates are considered equal

```
public static void main(String[] args) {  
    // create a point  
    Point2 p = new Point2(-1.0f, 1.5f);  
  
    // get its coordinates  
    System.out.println("p = " + p.toString());  
  
    // make a copy of p  
    Point2 q = new Point2(p);  
  
    // equals? yes!  
    System.out.println("p.equals(q) is: " + p.equals(q));  
}
```

true

The **equals** Contract

- ▶ for reference values **equals** is
 1. reflexive
 2. symmetric
 3. transitive
 4. consistent
 5. must not throw an exception when passed **null**

The `equals` contract: Reflexivity

1. reflexive :

- ▶ an object is equal to itself
- ▶ `x.equals(x)` is **true**

The `equals` contract: Symmetry

2. symmetric :

- ▶ two objects must agree on whether they are equal
- ▶ `x.equals(y)` is **true** if and only if `y.equals(x)` is **true**

The `equals` contract: Transitivity

3. transitive :

- ▶ if a first object is equal to a second, and the second object is equal to a third, then the first object must be equal to the third
- ▶ if
 - `x.equals(y)` is true**
 - and
 - `y.equals(z)` is true**
 - then
 - `x.equals(z)` must be true**

The `equals` contract: Consistency

4. **consistent :**
 - ▶ repeatedly comparing two objects yields the same result
(assuming the state of the objects does not change)

The `equals` contract: Non-nullity

5. `x.equals(null)` is always `false` and never throws an exception

Counter.equals

- ▶ the **Counter** version of equals returns true if the compared to object is a counter that has the same value as this counter

```
/**  
 * Compares this counter to the specified object. The result is {@code true}  
 * if and only if the argument is not {@code null} and is a {@code Counter}  
 * object that has the same current value as this object.  
 *  
 * @param obj  
 *         the object to compare this counter against  
 * @return true if the given object represents a Counter with the same  
 *         current value to this counter, false otherwise  
 */  
  
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj == null) {  
        return false;  
    }  
    if (this.getClass() != obj.getClass()) {  
        return false;  
    }  
    Counter other = (Counter) obj;  
    return this.value == other.value;  
}
```



equals using instanceof

Another version of `equals`

- ▶ the version of `equals` from the last lecture is perfectly fine if the class is never used as part of an inheritance hierarchy
 - ▶ we talk about inheritance later in the course
- ▶ otherwise, you should consider using the version of `equals` discussed in today's lecture
 - ▶ same as the version in the notebooks
 - ▶ see *Effective Java, Item 10* by Joshua Bloch

CISC124/CMPE212 Requirements

1. an instance is equal to itself
2. an instance is never equal to **null**
3. today's lecture:
 - ▶ instances with compatible types can be equal
4. instances with the same state are equal

1. An Instance is Equal to Itself

- ▶ `x.equals(x)` should always be `true`
- ▶ also, `x.equals(y)` should always be true if `x` and `y` are references to the same object
- ▶ you can check if two references are equal using `==`

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
}
```

2. An Instance is Never Equal to `null`

- ▶ Java requires that `x.equals(null)` returns `false`
- ▶ and you must not throw an exception if the argument is `null`
- ▶ unlike the previous lecture, we do not need to test for `null` because we get this test for free in the next step

3. Instances of compatible types can be equal

- ▶ `ref instanceof someReferenceType`
returns a boolean value
- ▶ the **instanceof** operator tests if a reference refers to an object that:
 - ▶ implements a specified interface, or
 - ▶ is an instance of the specified type, or
 - ▶ similar to the **getClass** test we used last lecture
 - ▶ is an instance of a subclass of the specified type
 - ▶ we will discuss this later in the course
- ▶ if the reference is **null** then **instanceof** returns **false**

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (!(this instanceof Point2)) {  
        return false;  
    }  
}
```

Instances with Same State are Equal

- ▶ recall that the value of the fields of an object define the state of the object
- ▶ two instances are equal if all* of their fields are equal
- ▶ unfortunately, we cannot yet retrieve the attributes of the parameter **obj** because it is declared to be an **Object** in the method signature
- ▶ we need a cast

* not necessarily all of the fields; there are many examples where only some of the fields are used to determine equality

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (!(this instanceof Point2)) {  
        return false;  
    }  
    Point2 other = (Point2) obj;  
}  
}
```

Instances with Same State are Equal

- ▶ there is a recipe for checking equality of fields
 - 1. if the field is a primitive type other than **float** or **double** use **==**
 - 2. if the field type is **float** use
Float.floatToIntBits
 - 3. if the attribute type is **double** use
Double.doubleToLongBits
 - 4. if the field is an array consider **Arrays.equals**
 - 5. if the field is a reference type use **equals**, but beware of fields that might be null

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(this instanceof Point2)) {
        return false;
    }
    Point2 other = (Point2) obj;
    if (Double.doubleToLongBits(this.x) != Double.doubleToLongBits(other.x)) {
        return false;
    }
    if (Double.doubleToLongBits(this.y) != Double.doubleToLongBits(other.y)) {
        return false;
    }
    return true;
}
```





hashCode

hashCode

- ▶ if you override **equals** you *must* override **hashCode**
 - ▶ otherwise, the hashed containers won't work properly
 - ▶ recall that we did not override **hashCode** for **Point2**

```
// client code somewhere
Point2 p = new Point2(1.0, -2.0);

HashSet<Point2> h = new HashSet<>();
h.add(p);
System.out.println( h.contains(p) );           // true

Point2 q = new Point2(1.0, -2.0);
System.out.println( h.contains(q) );           // false!
```

Lists as Containers

- ▶ suppose you have a list of unique **Point2** points
 - ▶ how do you compute whether or not the list contains a particular point?
 - ▶ write a loop to examine every element of the list

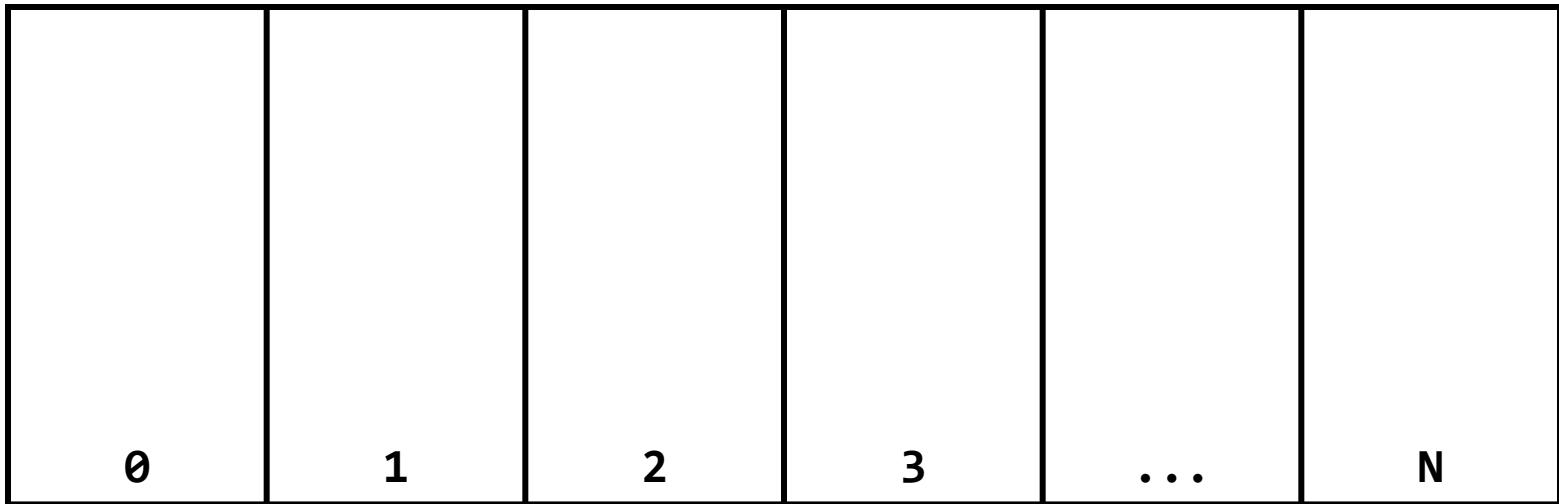
```
public static boolean
    hasPoint(Point2 p, List<Point2> points) {

    for ( Point2 point : points ) {
        if (point.equals(p)) {
            return true;
        }
    }
    return false;
}
```

- ▶ called *linear search* or *sequential search*
 - ▶ doubling the length of the array doubles the amount of searching we need to do
- ▶ if there are n elements in the list:
 - ▶ best case
 - ▶ the first element is the one we are searching for
 - 1 call to **equals**
 - ▶ worst case
 - ▶ the element is not in the list
 - n calls to **equals**
 - ▶ average case
 - ▶ the element is somewhere in the middle of the list
 - approximately $(n/2)$ calls to **equals** (in $O(n)$)

Hash Tables

- ▶ you can think of a hash table as being an array of buckets where each bucket holds the stored objects



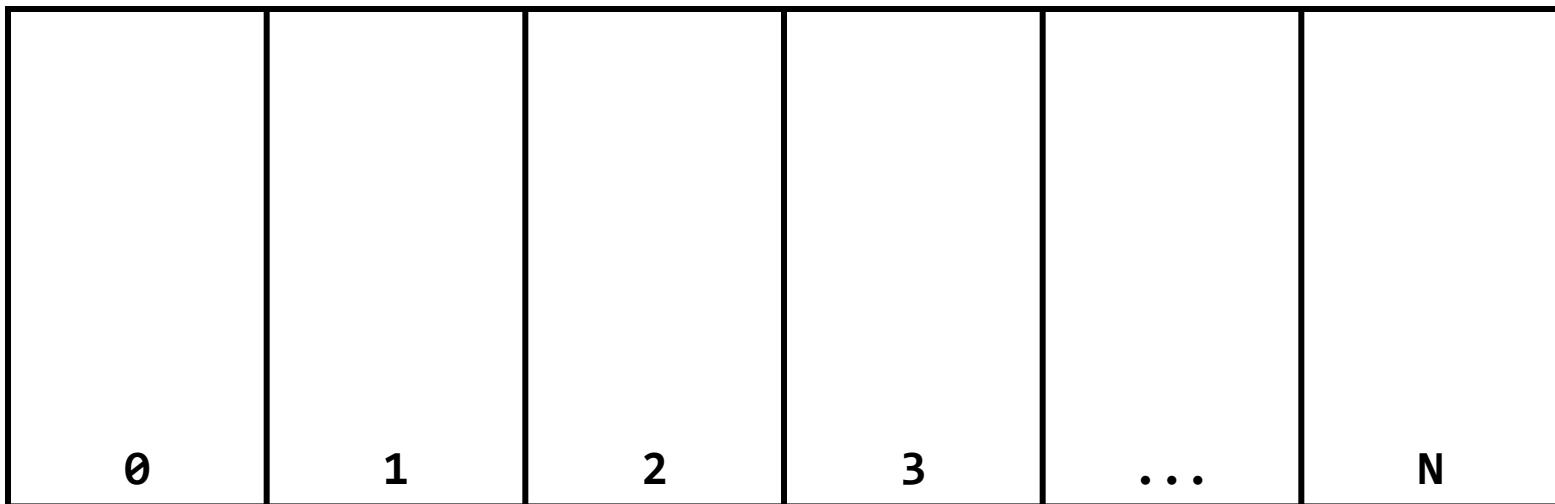
Insertion into a Hash Table

- to insert an object **a**, the hash table calls **a.hashCode()** method to compute which bucket to put the object into

b.hashCode() → 0

c.hashCode() ↗ N
d.hashCode() ↗ N

a.hashCode() → 2



means the hash table takes the hash code and does something to it to make it fit in the range **0–N**

Insertion into a Hash Table

- ▶ ideally, the distribution of elements in a hash table is uniform
- ▶ each bucket holds approximately the same number of elements

Search on a Hash Table

- to see if a hash table contains an object **a**, the hash table calls **a.hashCode()** method to compute which bucket to look for **a** in

z.hashCode() → N **a.hashCode() → 2**

b	a.equals(a) true			z.equals(c) z.equals(d) false	
0	1	2	3	...	N

- ▶ searching a hash table is usually much faster than linear search
 - ▶ doubling the number of elements in the hash table usually does not noticeably increase the amount of search needed
- ▶ if there are n elements in the hash table:
 - ▶ best case
 - ▶ the bucket is empty, or the first element in the bucket is the one we are searching for
 - 0 or 1 call to **equals**
 - ▶ worst case
 - ▶ all n of the elements are in the same bucket
 - n calls to **equals**
 - ▶ average case
 - ▶ the element is in a bucket with a small number of other elements
 - a small number of calls to **equals** (in $O(1)$)

Object.hashCode

- ▶ if you don't override **hashCode**, you get the implementation from **Object.hashCode**
- ▶ **Object.hashCode** uses the memory address of the object to compute the hash code

```

// client code somewhere
Point2 p = new Point2(1.0, -2.0);

HashSet<Point2> h = new HashSet<>();
h.add(p);
System.out.println( h.contains(p) );           // true

Point2 q = new Point2(1.0, -2.0);
System.out.println( h.contains(q) );           // false!

```

- ▶ note that **p** and **q** refer to distinct objects
 - ▶ therefore, their memory locations must be different
 - ▶ therefore, their hash codes are different (probably)
 - ▶ therefore, the hash table looks in the wrong bucket (probably)
 and does not find the complex number even though **p.equals(q)** is **true**

Implementing hashCode

- ▶ the basic idea is to generate a hash code using the fields of the object
 - ▶ see the recipe in the notebook
- ▶ it would be nice if two distinct objects had two distinct hash codes
 - ▶ but this is not required; two different objects can have the same hash code
- ▶ it is required that:
 1. if `x.equals(y)` then `x.hashCode() == y.hashCode()`
 2. `x.hashCode()` always returns the same value if `x` does not change its state

A bad (but legal) hashCode

```
public class Point2 {  
    public float x;  
    public float y;  
  
    @Override  
    public int hashCode() {  
        return 1;  
    }  
}
```

Don't do this, every point has the same hash code which causes a hashed collection to put every point in the same bucket.

Putting every object in the same bucket causes the hashed collection to behave like a list.

A slightly better (but bad) hashCode

```
public class Point2 {  
    public float x;  
    public float y;  
  
    @Override  
    public int hashCode() {  
        return (int) (this.x + this.y);  
    }  
}
```

Don't do this, similar points have the same hash code which causes a hashed collection to put these points in the same bucket.

A good hashCode

```
public class Point2x {  
    public float x;  
    public float y;  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(this.x, this.y);  
    }  
}
```

eclipse hashCode

- ▶ eclipse will also generate a hashCode method for you
 - ▶ Source → Generate hashCode() and equals()...
- ▶ it uses an algorithm that
 - ▶ “... yields reasonably good hash functions, [but] does not yield state-of-the-art hash functions, nor do the Java platform libraries provide such hash functions as of release 1.6. Writing such hash functions is a research topic, best left to mathematicians and theoretical computer scientists.”
 - ▶ Joshua Bloch, *Effective Java 2nd Edition*



compareTo



Comparable Objects

- ▶ many value types have a natural ordering
 - ▶ that is, for two objects **x** and **y**, **x** is less than **y** is meaningful
 - ▶ **Short**, **Integer**, **Float**, **Double**, etc
 - ▶ **Strings** can be compared in dictionary order
 - ▶ **Dates** can be compared in chronological order
 - ▶ you might compare points by their distance from the origin
- ▶ if your class has a natural ordering, consider implementing the **Comparable** interface
 - ▶ doing so allows clients to sort arrays or **Collections** of your object

Interfaces

- ▶ an interface is (usually) a group of related methods with empty bodies
- ▶ the **Comparable** interface has just one method

```
public interface Comparable<T>
{
    int compareTo(T t);
}
```

- ▶ a class that implements an interfaces promises to provide an implementation for every method in the interface

compareTo()

- ▶ Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
- ▶ Throws a **ClassCastException** if the specified object type cannot be compared to this object

Point2 compareTo

- ▶ suppose that we want to compare points by their distance from the origin; what steps do you need to implement **compareTo** for points?

```
public int compareTo(Point2 other)
```

1. compute the distance from the origin for **this** point
2. compute the distance from the origin for the **other** point
3. compare the two distances
 1. return a positive integer if **this** point is farther from the origin
 2. return a negative integer if **this** point is closer to the origin
 3. return zero if the points are equidistant from the origin

Point2 compareTo

```
public class Point2 implements Comparable<Point2> {  
    // fields, constructors, methods...
```

```
@Override  
public int compareTo(Point2 other) {  
    double thisDist = Math.hypot(this.x, this.y);  
    double otherDist = Math.hypot(other.x, other.y);  
    if (thisDist > otherDist) {  
        return 1;  
    }  
    else if (thisDist < otherDist) {  
        return -1;  
    }  
    return 0;  
}
```

Point2 compareTo

- ▶ don't forget what you learned in previous courses
 - ▶ you should delegate work to well-tested components where possible
- ▶ for distances, we need to compare two **double** values
 - ▶ **java.lang.Double** has methods that do exactly this

Point2 compareTo

```
public class Point2 implements Comparable<Point2> {  
    // fields, constructors, methods...
```

```
@Override  
public int compareTo(Point2 other) {  
    double thisDist = Math.hypot(this.x, this.y);  
    double otherDist = Math.hypot(other.x, other.y);  
    return Double.compare(thisDist, otherDist);  
}
```

Comparable Contract

- ▶ **compareTo** has a contract that is described in documentation for the **Comparable** interface
- ▶ the contract ensures that **compareTo** defines a natural ordering

Consistency with equals

- ▶ an implementation of `compareTo()` is said to be consistent with `equals()` when

if `x.compareTo(y)` returns 0 then
`x.equals(y)` returns true

- ▶ and

if `x.equals(y)` returns true then
`x.compareTo(y)` returns 0

Not in the Comparable Contract

- ▶ it is *not* required that **compareTo()** be consistent with **equals()**
 - ▶ that is
 - if **x.compareTo(y)** returns **0** then
x.equals(y) can return **false**
 - ▶ similarly
 - if **x.equals(y)** returns **true** then
x.compareTo(y) can return a non-zero value
- ▶ try to come up with examples for both cases above

Implementing `compareTo`

- ▶ if you are comparing fields of type `float` or `double` you should use `Float.compare` or `Double.compare` instead of `<`, `>`, or `==` unless you decide that the result of `compareTo` should encode some sort of information
- ▶ if your `compareTo` implementation is broken, then any classes or methods that rely on `compareTo` will behave erratically
 - ▶ `TreeSet`, `TreeMap`
 - ▶ many methods in the utility classes `Collections` and `Arrays`

Counter `compareTo`

- ▶ sometimes the value returned by `compareTo` can encode additional information
- ▶ for example, the **Counter** version of **compareTo** returns the difference in the values of the two counters

```
public class Counter implements Comparable<Counter> {  
    // fields, constructors, other methods not shown  
  
    /**  
     * Compares the value of this counter to the value of  
     * another counter. Returns the difference between the  
     * value of this counter and the other counter.  
     *  
     * @param other  
     *         the other counter to compare to  
     * @return the difference between the value of this  
     *         counter and the other counter  
     */  
    @Override  
    public int compareTo(Counter other) {  
        return this.value - other.value;  
    }  
}
```

Enumerated Types

Motivation

- ▶ consider the following groups:
 - ▶ days of the week
 - ▶ months of the year
 - ▶ suits in a standard deck of playing cards
 - ▶ ranks in a standard deck of playing cards
 - ▶ Canadian coins
 - ▶ Canadian provinces and territories
 - ▶ planets of the solar system
 - ▶ arithmetic operations
- ▶ what do these groups have in common?

Enumerated types

- ▶ an *enumerated type* (or *enum type* or *enum*) is a type whose values consist of a fixed set of constants
 - ▶ Sunday, Monday, ..., Saturday
 - ▶ January, February, ..., December
 - ▶ clubs, diamonds, hearts, spades
 - ▶ 2, 3, ..., ace
 - ▶ nickel, dime, ..., toonie
 - ▶ and so on

Old Style Enums

- ▶ older Java code and C code used **int** constants to represent enumerated types

```
public static final int SUNDAY = 0;  
public static final int MONDAY = 1;  
public static final int TUESDAY = 2;  
// and so on
```

```
public static final int JANUARY = 0;  
public static final int FEBRUARY = 1;  
public static final int MARCH = 2;  
// and so on
```

Old Style Enums

- ▶ a major problem with using ints to represent enums is that there is no type safety
- ▶ any method with an int parameter will accept a day or month
- ▶ there is no way to restrict a parameter to be only a day or a month
- ▶ you can perform arithmetic with days and months
- ▶ you can compare days and months (for equality, inequality, less than, greater than)

Old Style Enums

- ▶ another problem with using ints to represent enums is that there is no easy way to translate the int value to a string
- ▶ there is no **toString** method for ints

Old Style Enums

- ▶ for enumerations such as months or days it is also common to use strings for the constants instead of ints

```
public static final String SUNDAY = "SUNDAY";
public static final String MONDAY = "MONDAY";
public static final String TUESDAY = "TUESDAY";
// and so on
```

```
public static final String JANUARY = "JANUARY";
public static final String FEBRUARY = "FEBRUARY";
public static final String MARCH = "MARCH";
// and so on
```

Old Style Enums

- ▶ using strings for the constants isn't much better than using ints
- ▶ all of your enumerations are now strings so there is still no real type safety
- ▶ it is tempting to use "**SUNDAY**" instead of **Month.SUNDAY**
- ▶ comparing strings for equality is slower than comparing ints or addresses

Java enums

- ▶ Java 1.5 added an enum type
- ▶ enums are true Java classes that implicitly inherit from the superclass `java.lang.Enum`
- ▶ enums can have fields and methods and can implement interfaces
- ▶ enums can have constructors but the constructors are not accessible outside of the enum
 - ▶ i.e., the constructors are implicitly **private**

Java enums

- ▶ an example of a simple enum:

```
public enum Day {  
    SUNDAY,  
    MONDAY,  
    TUESDAY,  
    WEDNESDAY,  
    THURSDAY,  
    FRIDAY,  
    SATURDAY;  
}
```

Java enums

- ▶ an example of using the **Day** enum:

```
Day d = Day.THURSDAY;  
System.out.println(d);
```

- ▶ prints:

THURSDAY

- ▶ notice that enums provide a compiler generated **toString** method
 - ▶ you can override **toString** if you wish

Java enums

- ▶ an enum exports exactly one instance for each enumeration constant via a public static final field
- ▶ this means that you can safely compare enumeration references using `==` or `!=`
 - ▶ can also use `equals`
- ▶ because there are no accessible constructors there can be no instances except the ones exported by the enum

Type safety

- ▶ enums are types; therefore you get all of the benefits of compile-time type safety
- ▶ suppose you have a second enum type:

```
public enum Month {  
    JANUARY, FEBRUARY,  
    MARCH, APRIL,  
    MAY, JUNE,  
    JULY, AUGUST,  
    SEPTEMBER, OCTOBER,  
    NOVEMBER, DECEMBER;  
}
```

Enums implement Comparable

- ▶ enums automatically implement the **Comparable** interface
- ▶ the natural ordering is the order in which the constants are defined
 - ▶ unfortunately, there is no way for the implementer to override the behavior of **compareTo** in an enum

Enums implement Comparable

- ▶ example using Month:

```
Day d1 = Day.MONDAY;
Day d2 = Day.THURSDAY;
Day d3 = Day.SATURDAY;
System.out.println(d1.compareTo(d2));
System.out.println(d3.compareTo(d1));
System.out.println(d2.compareTo(d2));
```

- ▶ prints:

-3
5
0

Enums can have fields

- ▶ enums can have fields and the fields can be initialized via a constructor
- ▶ but remember that constructors for enums are always private
- ▶ enums are supposed to be constants

Enums can have fields

```
public enum Month {  
    JANUARY(31),  
    FEBRUARY(28),  
    MARCH(31),  
    APRIL(30),  
    MAY(31),  
    JUNE(30),  
    JULY(31),  
    AUGUST(31),  
    SEPTEMBER(30),  
    OCTOBER(31),  
    NOVEMBER(30),  
    DECEMBER(31);  
  
    private final int days;
```

calls **Month** constructor

Enums can have fields

```
Month(int days) {  
    this.days = days;  
}
```

constructor is **private** (illegal
to add an access modifier that
is not **private**)

```
public int days(int year) {  
    if (this != FEBRUARY) {  
        return this.days;  
    }  
    if (year % 400 == 0 ||  
        (year % 4 == 0 && year % 100 != 0)) {  
        return this.days + 1;  
    }  
    return this.days;  
}
```

`!=` ok because there is only
one instance of each constant

The values method

- ▶ every enum has a compiler generated public static method called **values**
- ▶ **values** returns an array of the enumeration constants in the order that they were declared

The values method

- ▶ example using Month:

```
System.out.println(Arrays.toString(Month.values()));
```

- ▶ prints (all on one line):

```
[JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY,  
AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER]
```

The values method

- ▶ another example using Month:

```
System.out.println(Arrays.toString(Month.values()));
for (Month m : Month.values()) {
    System.out.println(m + " : " + m.days(2018));
}
```

The values method

- ▶ prints:

JANUARY : 31

FEBRUARY : 28

MARCH : 31

APRIL : 30

MAY : 31

JUNE : 30

JULY : 31

AUGUST : 31

SEPTEMBER : 30

OCTOBER : 31

NOVEMBER : 30

DECEMBER : 31

The `valueOf` method

- ▶ every enum has a compiler generated public static method with the signature **`valueOf(String)`**
- ▶ **`valueOf`** returns the constant that corresponds to the argument string

The valueOf method

- ▶ example using Month:

```
Month m = Month.valueOf("MAY");
System.out.println(m == Month.MAY);
```

- ▶ prints:

true

Constant-specific methods

- ▶ each constant in an enumeration can have its own version of a method
- ▶ to do this, you declare an abstract method in the enum
 - ▶ each constant then overrides the abstract method in a constant-specific class body
 - ▶ learn more about abstract methods when we discuss inheritance
- ▶ consider creating an enum that represents the different arithmetic operations of a calculator

see Effective Java 3rd Edition, Item 34

```
public enum Operation {  
    PLUS {  
        @Override  
        public double apply(double x, double y) { return x + y; }  
    },  
    MINUS {  
        @Override  
        public double apply(double x, double y) { return x - y; }  
    },  
    TIMES {  
        @Override  
        public double apply(double x, double y) { return x * y; }  
    },  
    DIVIDE {  
        @Override  
        public double apply(double x, double y) { return x / y; }  
    };  
  
    public abstract double apply(double x, double y);  
}
```

Prefer enums to Boolean parameters

- ▶ booleans are often used as parameters when there is a choice between two different values
- ▶ consider the following hypothetical constructors:
 - ▶ **BinaryDigit(boolean value)**
 - ▶ true for 1, false for 0
 - ▶ **Thermometer(boolean scale)**
 - ▶ true for degrees Celcius, false for degrees Fahrenheit

Prefer enums to Boolean parameters

- ▶ using enums instead of boolean values makes your code much more readable
- ▶ `new BinaryDigit(BinaryValue.ONE)` or
`new BinaryDigit(BinaryValue.ZERO)`
- ▶ `new Thermometer(TemperatureScale.CELCIUS)` or
`new Thermometer(TemperatureScale.FAHRENHEIT)`

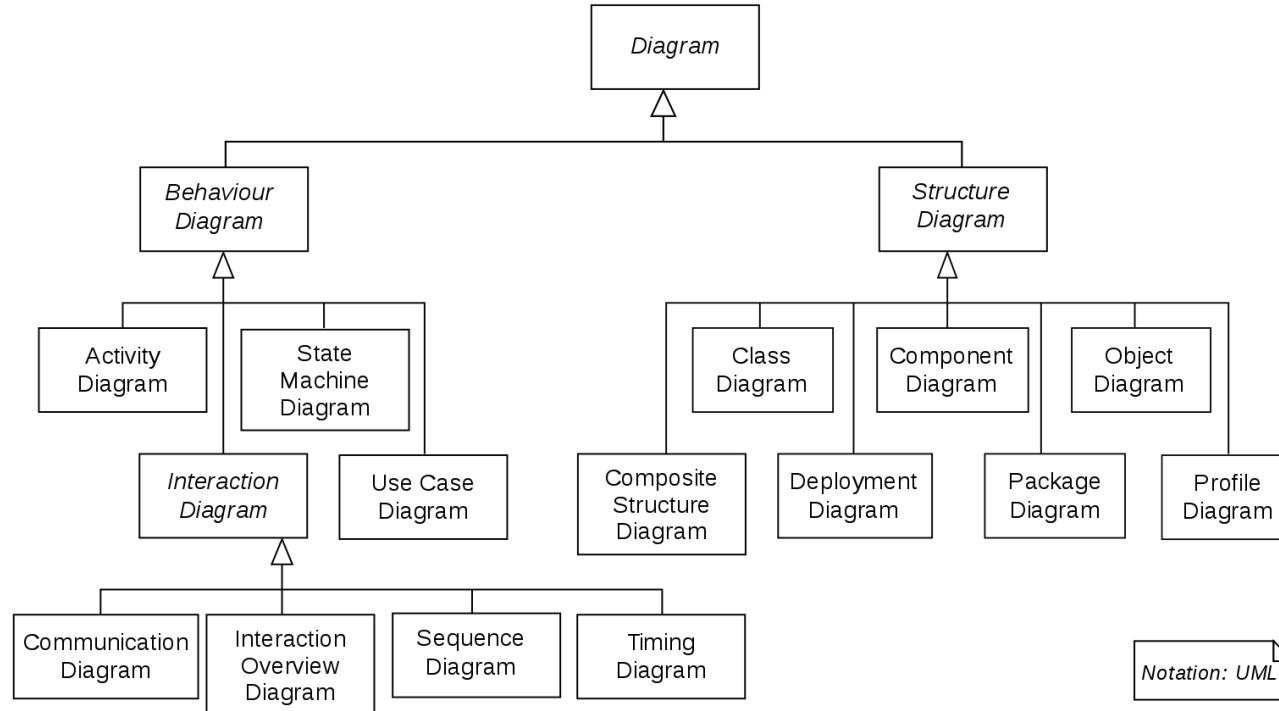
When to use an enum

- ▶ use an enum when you need a set of constants whose values are known at compile time
- ▶ compare the **Card** class implemented using **String** versus the **Card** class implemented using enums

UML class diagrams

Unified Modelling Language (UML)

- ▶ a visual modelling language used in software engineering for visualizing the design of a software system



UML class diagram

- ▶ a UML class diagram describes the structure of a class
- ▶ includes the name of the class and possibly:
 - ▶ fields
 - ▶ UML calls these the attributes of the class
 - ▶ methods
 - ▶ also called the operations of the class

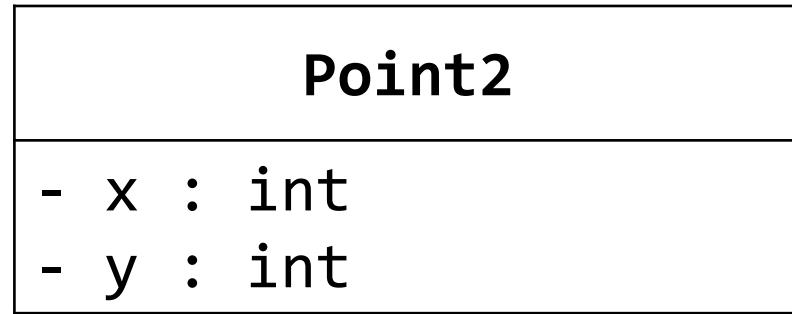
UML class diagram

- ▶ the simplest class diagram is just the name of the class in a box
- ▶ this is the only mandatory information
- ▶ class name is centered and capitalized

Point2

UML class diagram

- ▶ the fields of the class can be shown in the next compartment below the name
- ▶ access modifier, followed by field name, followed by field type
- ▶ text is left justified



UML class diagram

- ▶ access modifiers for fields
 - ▶ + public
 - ▶ - private
 - ▶ ~ package private
 - ▶ # protected

UML class diagram

- ▶ the methods of the class can be shown in the next compartment below the fields
- ▶ access modifier, followed by method name, followed by parameter list, followed by return type
 - ▶ parameters are specified as *name* : *type*
- ▶ constructors can be included as well
- ▶ text is left justified

Point2

- x : int
- y : int
- + Point2()
- + Point2(x : double, y : double)
- + Point2(other : Point2)
- + x() : double
- + y() : double
- + x(x : double) : Point2
- + y(y : double) : Point2
- + equals(obj : Object) : boolean

and so on

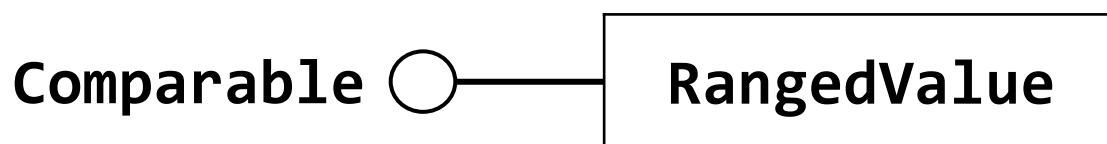
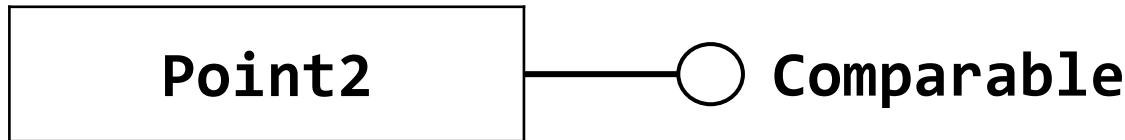
RangedValue

- value : double
- range : Range
- + RangedValue(min : double, max : double,
 value : double)
- + min() : double
- + max() : double
- + range() : Range
- + value() : double

and so on

Classes that implement an interface

- ▶ both **Point2** and **RangedValue** implement the **Comparable** interface which can be indicated using the "lollipop" notation



Relationships between classes

Relationships between classes

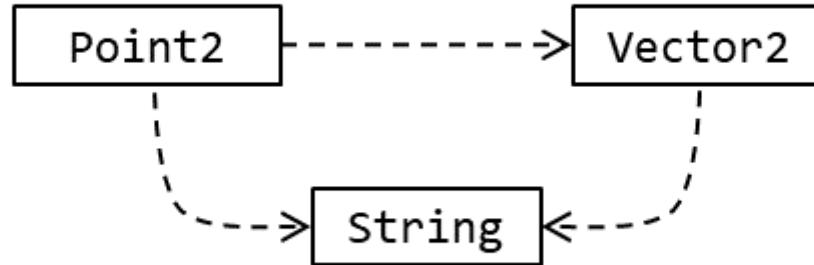
- ▶ an object-oriented program is made up of multiple interacting classes
- ▶ the relationships between the classes is important
 - ▶ classes are built using other classes (structure)
 - ▶ classes interact with other classes (behavior)

Dependency

- ▶ dependency is the weakest relationship between classes
 - ▶ models the *uses* relationship
- ▶ if a class **X** uses another class **Y**, then we say that **X** depends on
 - ▶ it can also be the case that **Y** depends on **X**
 - ▶ dependency can be a two-way relationship
- ▶ **X** depends on **Y** if **Y** appears as:
 - ▶ a field of **X**
 - ▶ a parameter in a constructor or method of **X**
 - ▶ a variable type in a constructor or method of **X**
 - ▶ a return type of a method of **X**

UML class diagram

- dependency can be shown in a UML class diagram using a dashed line ending in an open arrow that points to the class depended on
- in the diagram below:
 - Point2** depends on **Vector2** and **String**
 - Vector2** depends on **String**

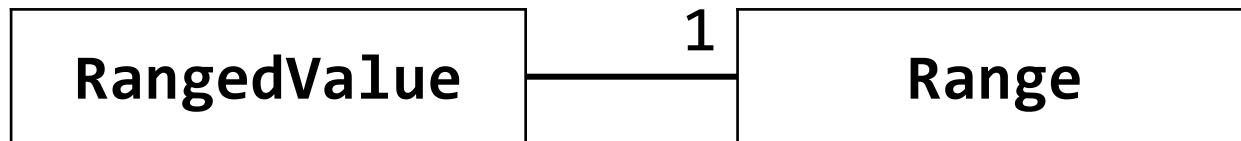


Association

- ▶ association is a stronger form of dependency
- ▶ models the *has-a* relationship
- ▶ if a class **X** has a field of type **Y**, then we say that **X** has an association with **Y**
- ▶ it can also be the case that **Y** has an association with **X**
 - ▶ association can be a two-way relationship

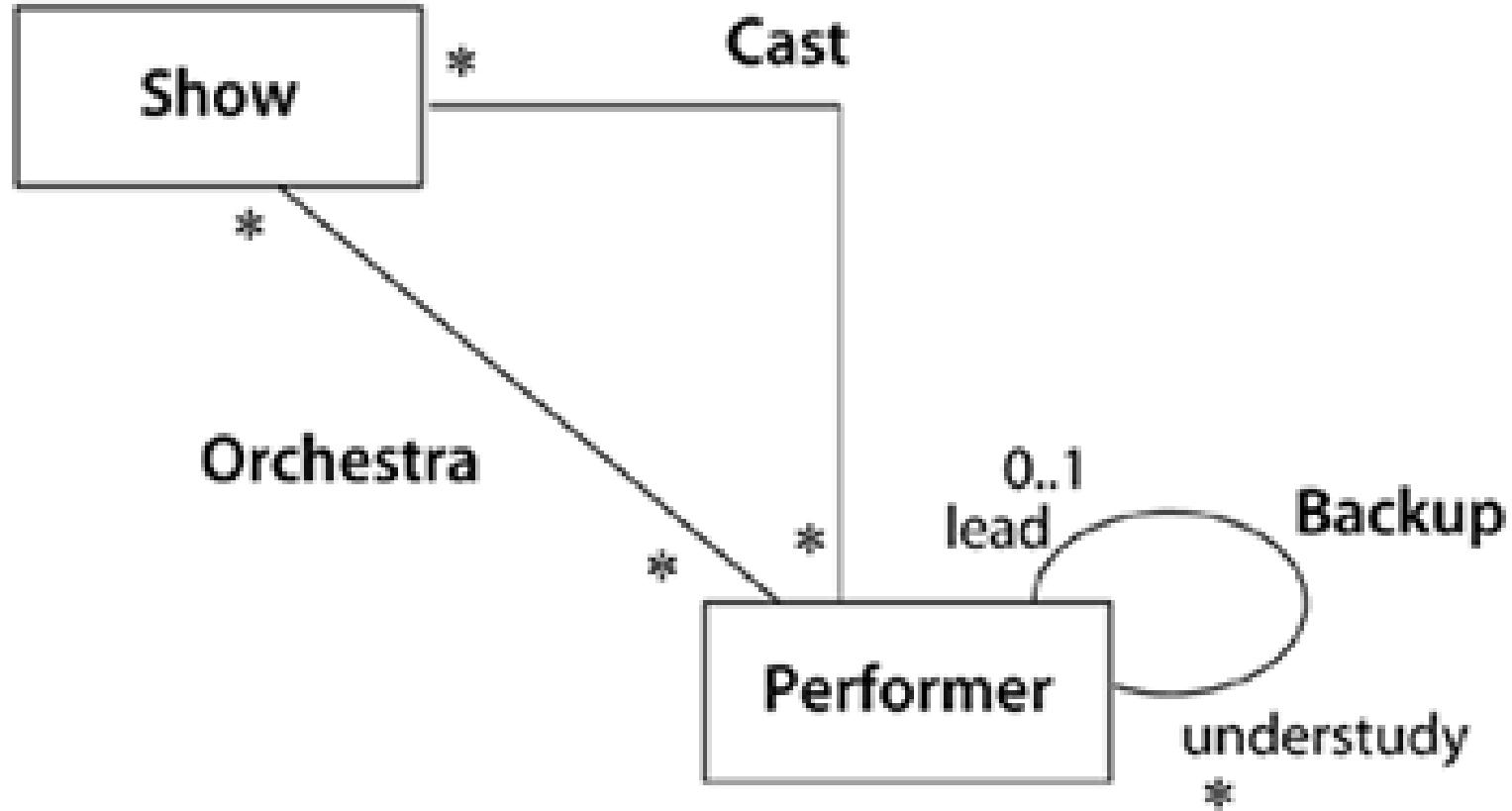
UML class diagram

- ▶ association can be shown in a UML class diagram using a solid line connecting the two classes
- ▶ in Assignment 3, every **RangedValue** object has one **Range** object
 - ▶ we can indicate the number of Range objects that participate in the relationship using a *multiplicity element*



* the relationship between **RangedValue** and **Range** is actually stronger than association

UML class diagram



Aggregation and Composition

Aggregation and Composition

- ▶ aggregation and composition are stronger forms of association
- ▶ the terms aggregation and composition are used to describe a relationship between objects
- ▶ both terms describe the *has-a* relationship (like association)
 - ▶ the university has-a collection of departments
 - ▶ each department has-a collection of professors

Aggregation and Composition

- ▶ composition implies ownership
 - ▶ if the university disappears then all of its departments disappear
 - ▶ a university is a *composition* of departments
- ▶ aggregation does not imply ownership
 - ▶ if a department disappears then the professors do not disappear
 - ▶ a department is an *aggregation* of professors
- ▶ both aggregation and composition are one-way relationships

Aggregation

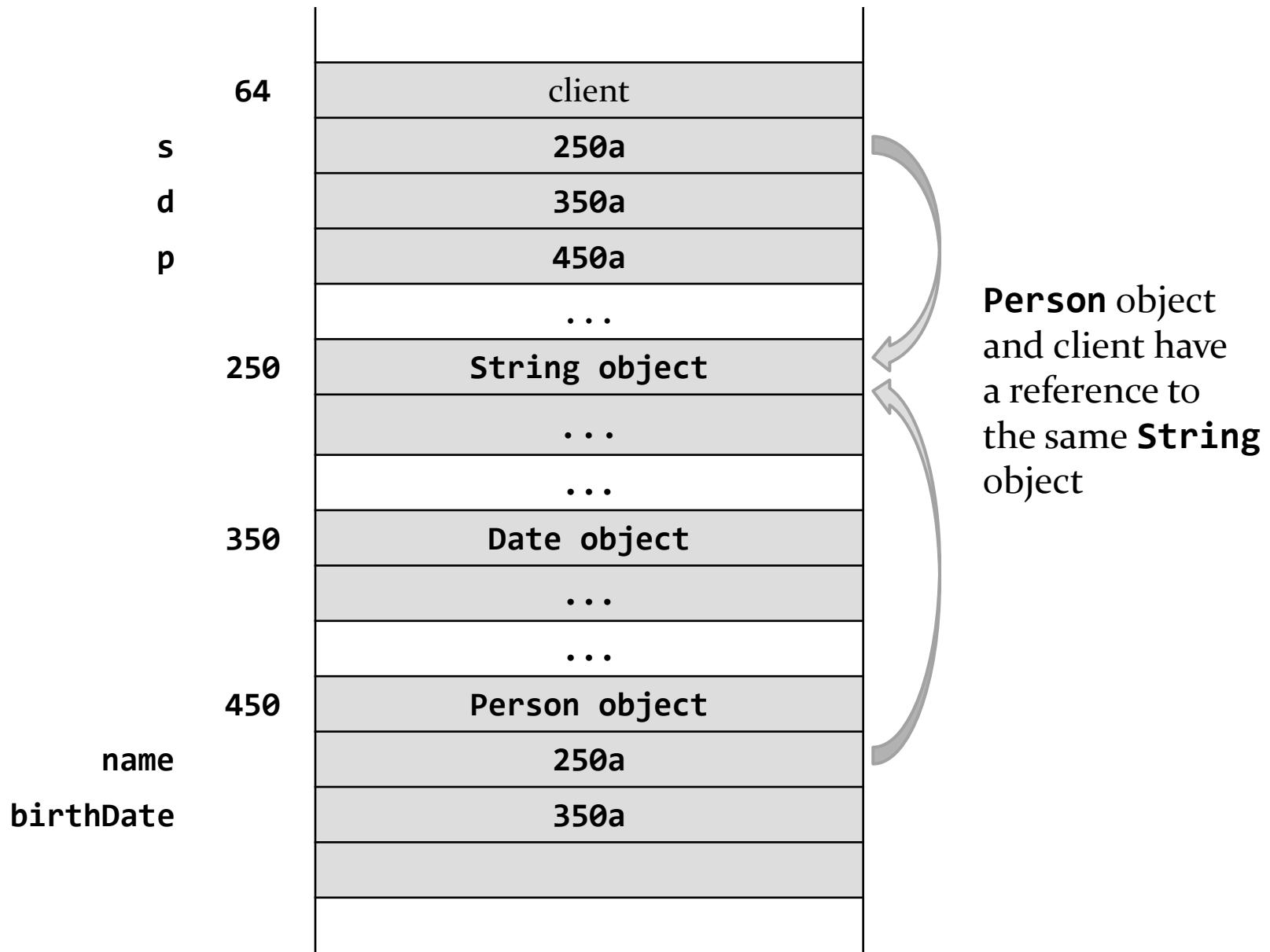
- ▶ suppose a **Person** has a name and a date of birth

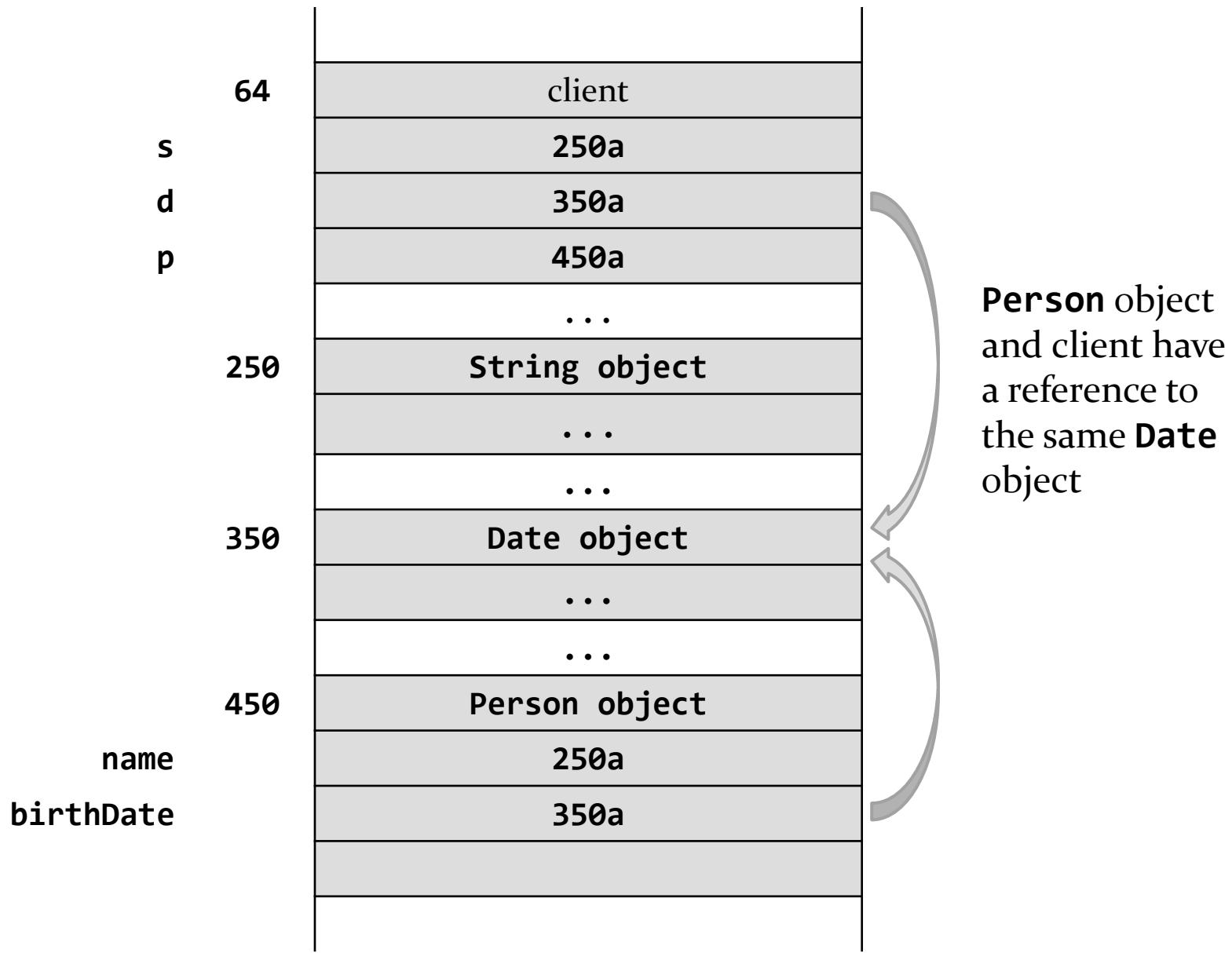
```
public class Person {  
    private String name;  
    private Date birthDate;
```

```
public Person(String name, Date birthDate) {  
    this.name = name;  
    this.birthDate = birthDate;  
}  
  
public String getName() {  
    return this.name;  
}  
  
public Date getBirthDate() {  
    return this.birthDate;  
}  
}
```

- ▶ the **Person** example uses aggregation
 - ▶ notice that the constructor does not make a new copy of the name and birth date objects passed to it
 - ▶ the name and birth date objects are shared with the client
 - ▶ both the client and the **Person** instance are holding references to the same name and birth date

```
// client code somewhere
String s = "Billy Bob";
Date d = new Date(91, 2, 26); // March 26, 1991
Person p = new Person(s, d);
```





- ▶ what happens when the client modifies the **Date** instance?

```
// client code somewhere
String s = "Billy Bob";
Date d = new Date(90, 2, 26); // March 26, 1990
Person p = new Person(s, d);

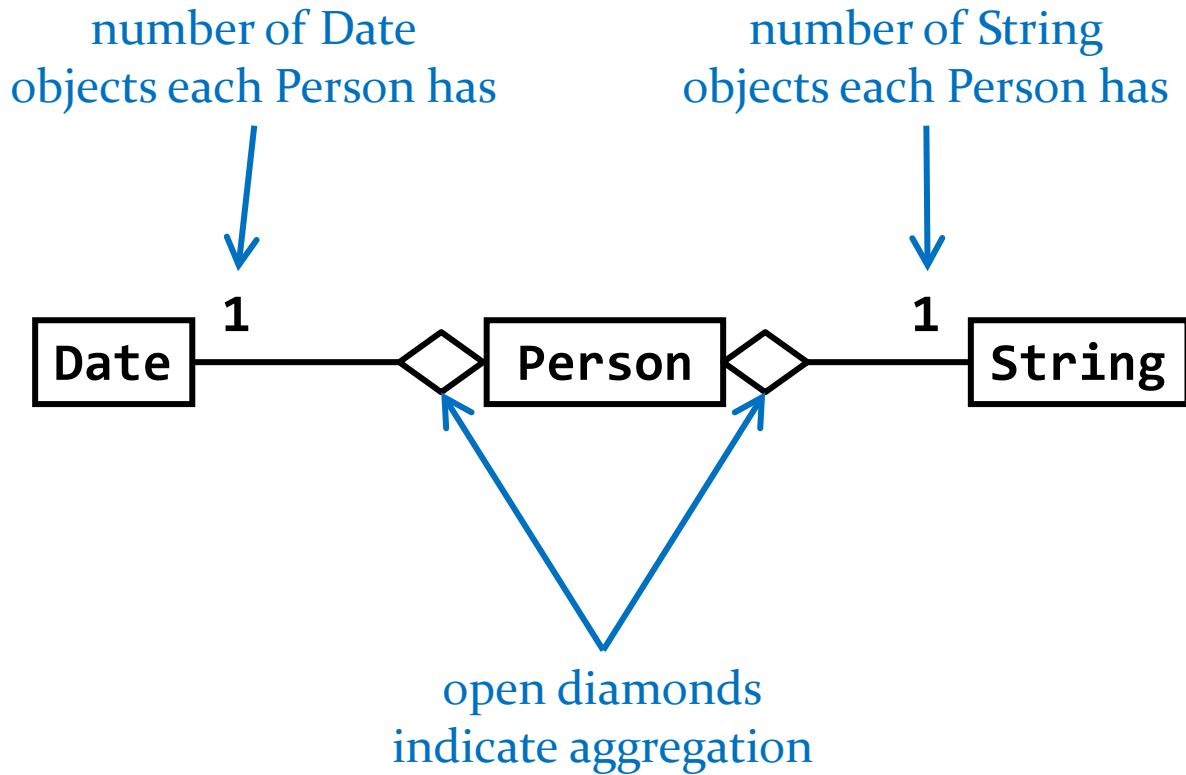
d.setYear(95);           // November 3, 1995
d.setMonth(10);
d.setDate(3);
System.out.println( p.getBirthDate() );
```

- ▶ prints **Fri Nov 03 00:00:00 EST 1995**

-
- ▶ because the **Date** instance is shared by the client and the **Person** instance:
 - ▶ the client can modify the date using **d** and the **Person** instance **p** sees a modified **birthDate**
 - ▶ the **Person** instance **p** can modify the date using **birthDate** and the client sees a modified date **d**

-
- ▶ note that even though the **String** instance is shared by the client and the **Person** instance **p**, neither the client nor **p** can modify the **String**
 - ▶ immutable objects make great building blocks for other objects
 - ▶ they can be shared freely without worrying about their state

UML Class Diagram for Aggregation



Another Aggregation Example

- ▶ consider implementing a projectile whose position is governed by the following equations of motion

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \mathbf{v}_i \delta t + \frac{1}{2} \mathbf{g} \delta t^2$$

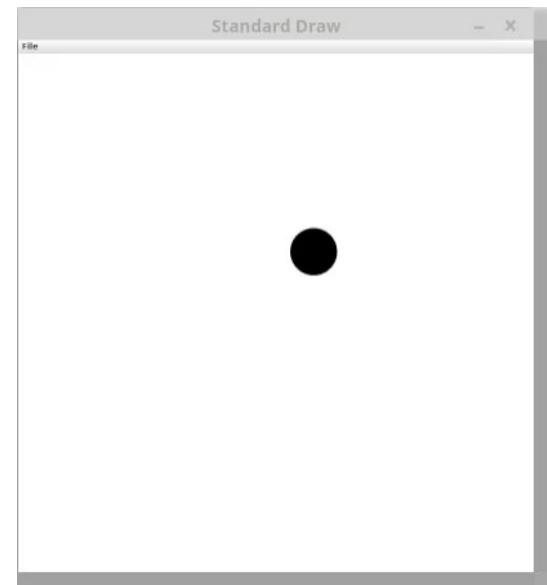
$$\mathbf{v}_{i+1} = \mathbf{v}_i + \mathbf{g} \delta t$$

\mathbf{p}_i position at time t_i

\mathbf{v}_i velocity at time t_i

\mathbf{g} acceleration due to gravity

$$\delta t = t_{i+1} - t_i$$



Another Aggregation Example

- ▶ the **Projectile** has-a **Point2** that represents the position of the ball and a **Vector2** that represents the velocity of the ball



```
package ca.queensu.cs.cisc124.notes.basics.geometry;  
/**  
 * A particle moving with approximate projectile motion in two  
 * dimensions. It is assumed that the only force acting on  
 * the projectile is gravity.  
 */  
public class Projectile {  
  
    /**  
     * The current position of the projectile.  
     */  
    private Point2 pos;  
  
    /**  
     * The current velocity of the projectile.  
     */  
    private Vector2 vel;
```

```
/**  
 * Initialize this projectile to start at position (0, 0)  
 * having velocity (0, 0).  
 *  
 */  
public Projectile() {  
    this.pos = new Point2();  
    this.vel = new Vector2();  
}
```

```
/**  
 * Returns a reference to the position of this projectile.  
 *  
 * @return a reference to the position of this projectile  
 */  
public Point2 getPosition() {  
    return this.position;  
}  
  
/**  
 * Returns a reference to the velocity of this projectile.  
 *  
 * @return a reference to the velocity of this projectile  
 */  
public Vector2 getVelocity() {  
    return this.velocity;  
}
```

```
/**  
 * Set the position of this projectile to the specified position.  
 * Returns the old position of this projectile.  
 *  
 * @param p the new position of this projectile  
 * @return the old position of this projectile  
 */  
public Point2 setPosition(Point2 p) {  
    Point2 oldPos = this.pos;  
    this.pos = p;  
    return oldPos;  
}
```

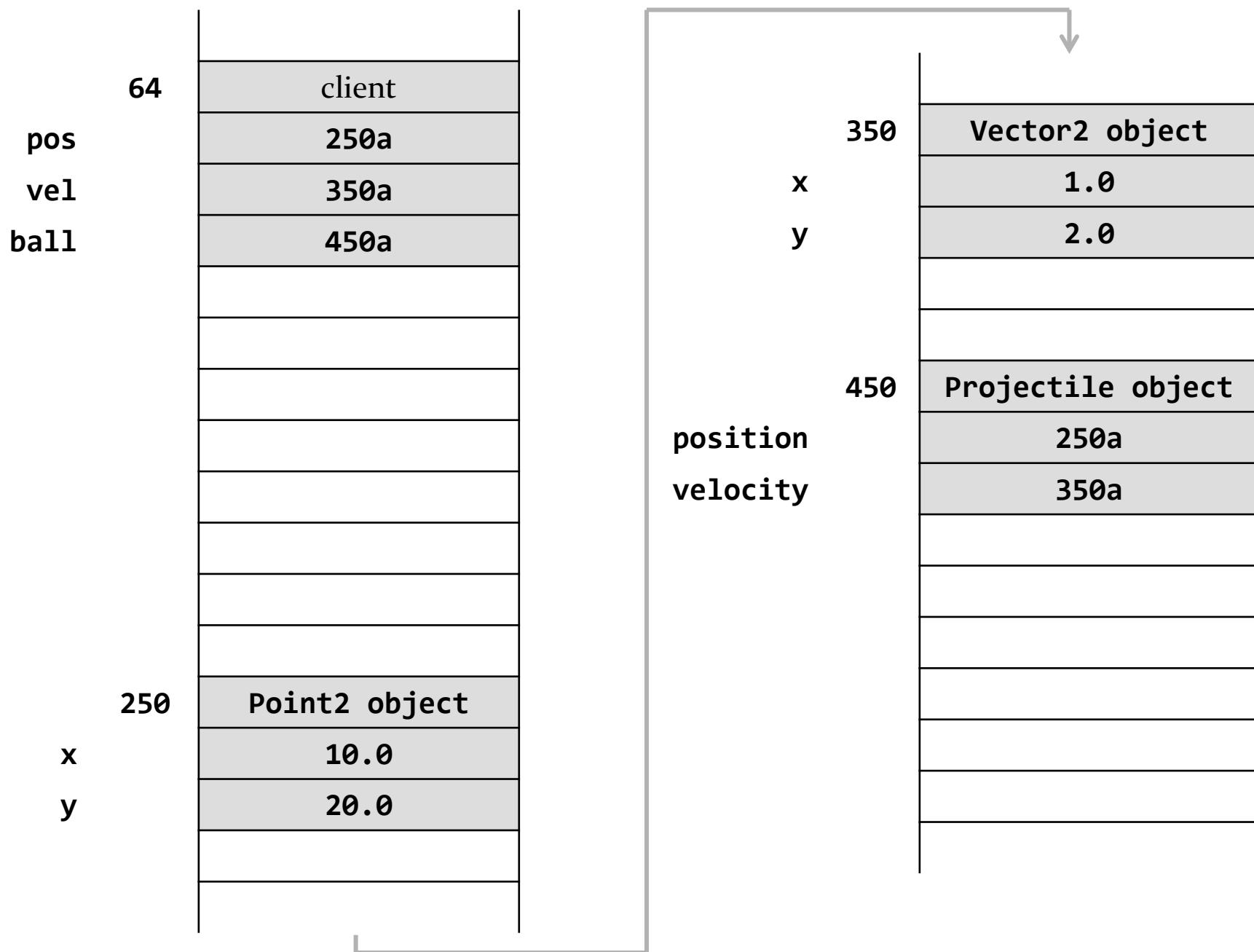
```
/**  
 * Set the position of this projectile to the specified velocity.  
 * Returns the old velocity of this projectile.  
 *  
 * @param p the new velocity of this projectile  
 * @return the old velocity of this projectile  
 */  
public Vector2 setVelocity(Vector2 v) {  
    Vector2 oldVel = this.vel;  
    this.vel = v;  
    return oldVel;  
}
```

```
/**  
 * Updates the position and velocity of this projectile after  
 * the projectile has moved {@code dt} seconds from its previous  
 * position.  
 *  
 * @param dt the time period over which the projectile has moved  
 */  
public void move(double dt) {  
    // acceleration due to gravity  
    Vector2 g = new Vector2();  
    g.set(0.0, -9.81);  
  
    this.pos.add(Vector2.multiply(dt, this.vel)).  
        add(Vector2.multiply(0.5 * dt * dt, g));  
    this.vel.add(Vector2.multiply(dt, g));  
}
```

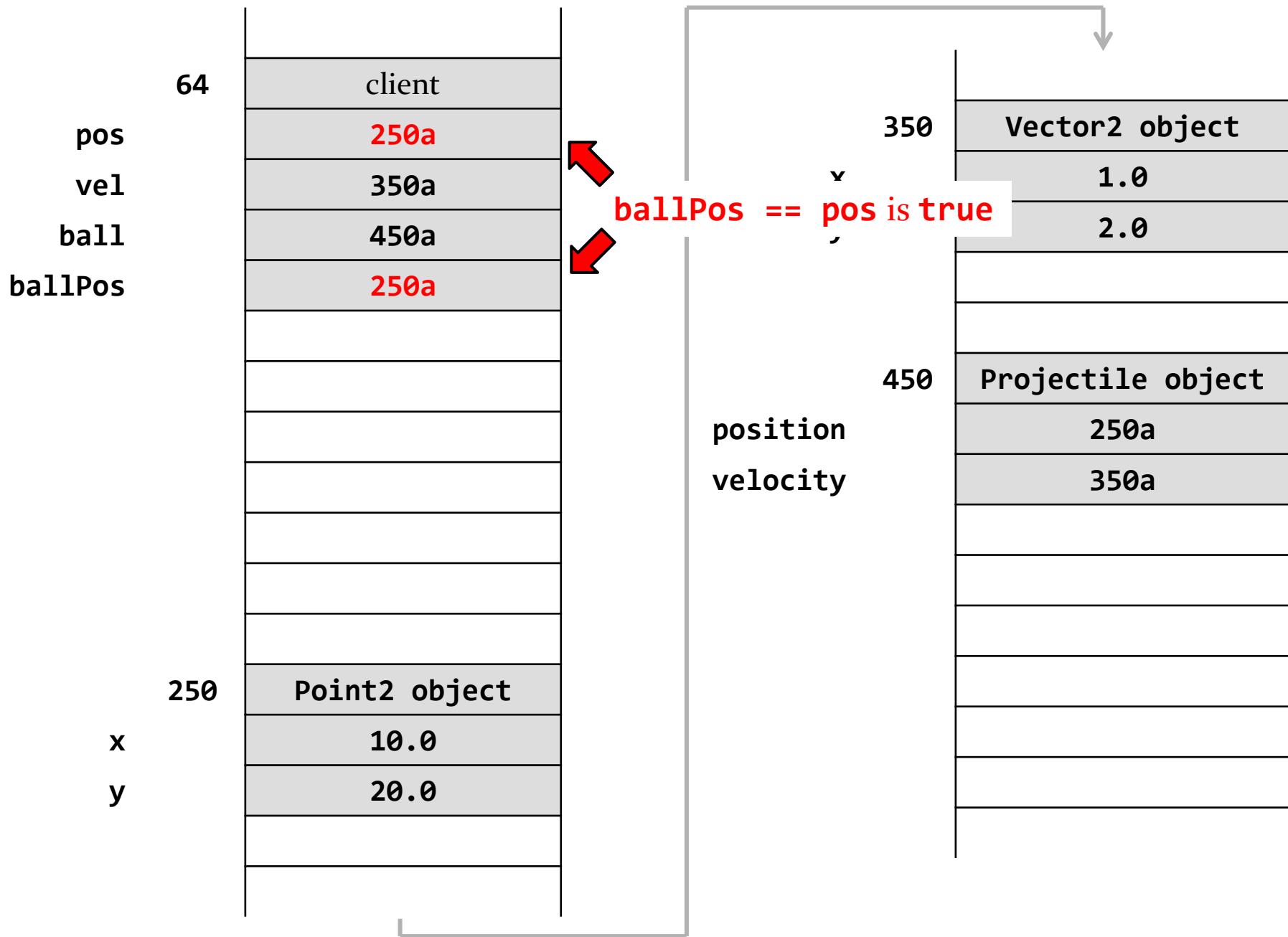
Projectile as an aggregation

- ▶ implementing **Projectile** is very easy
- ▶ fields
 - ▶ are references to existing objects provided by the client
- ▶ accessors
 - ▶ give clients a reference to the aggregated **Point2** and **Vector2** objects
- ▶ mutators
 - ▶ set fields to existing object references provided by the client
- ▶ we say that the **Projectile** fields are *aliases*

```
public static void main(String[] args) {  
    Point2 pos = new Point2(10.0, 20.0);  
    Vector2 vel = new Vector2(1.0, 2.0);  
    Projectile ball = new Projectile();  
    ball.setPosition(pos);  
    ball.setVelocity(vel);  
}
```

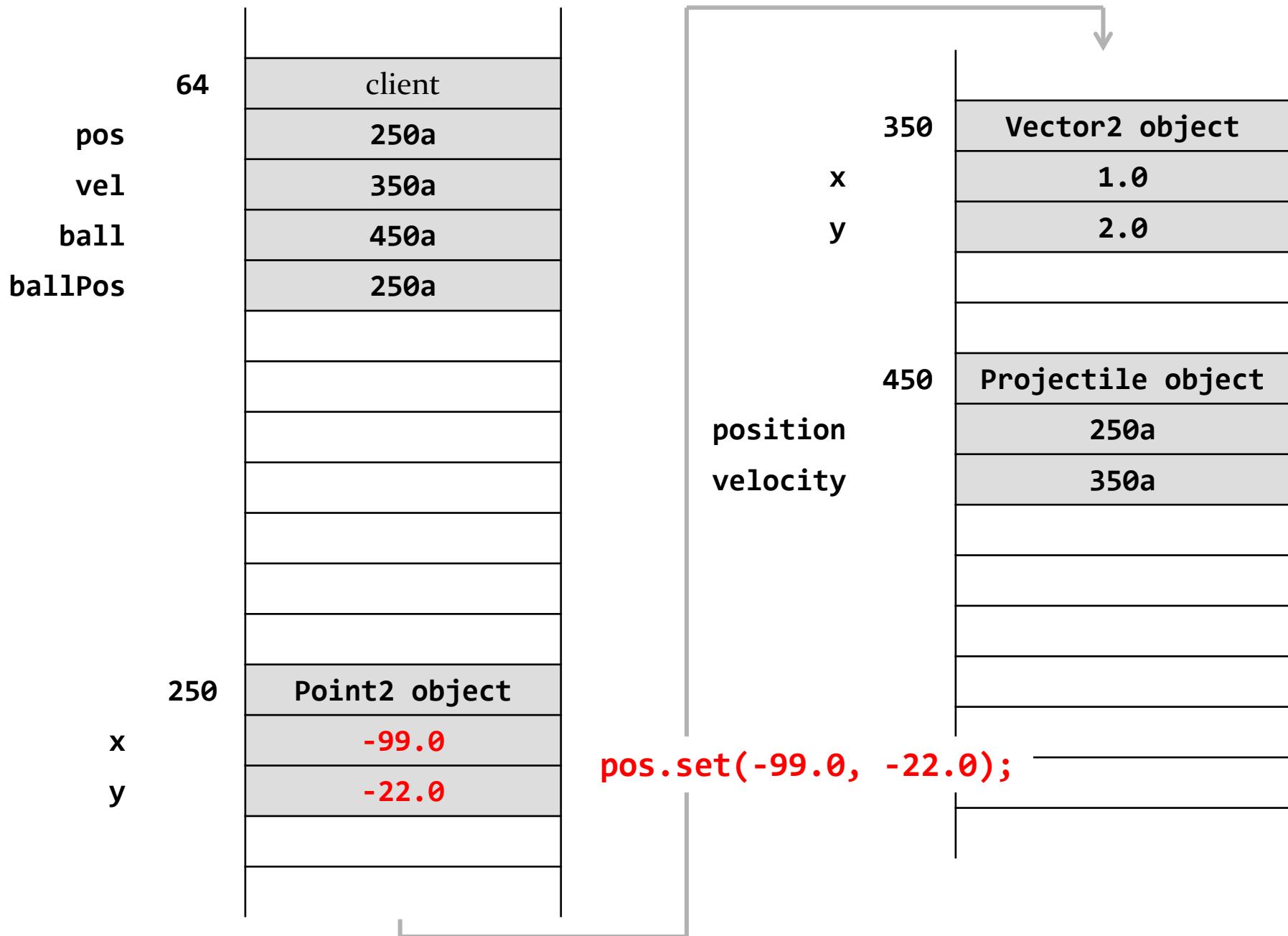


```
public static void main(String[] args) {  
    Point2 pos = new Point2(10.0, 20.0);  
    Vector2 vel = new Vector2(1.0, 2.0);  
    Projectile ball = new Projectile();  
    ball.setPosition(pos);  
    ball.setVelocity(vel);  
  
    // does ball and client share the same objects?  
    Point2 ballPos = ball.getPosition();  
    System.out.println("same Point2 object?: " + (ballPos == pos));  
}
```



```
public static void main(String[] args) {  
    Point2 pos = new Point2(10.0, 20.0);  
    Vector2 vel = new Vector2(1.0, 2.0);  
    Projectile ball = new Projectile();  
    ball.setPosition(pos);  
    ball.setVelocity(vel);  
  
    // does ball and client share the same objects?  
    Point2 ballPos = ball.getPosition();  
    System.out.println("same Point2 object?: " + (ballPos == pos));  
  
    // client changes pos  
    pos.set(-99.0, -22.0);  
    System.out.println("ball position: " + ballPos);  
}
```

"ball position: (-99.0, -22.0)"



Projectile as aggregation

- ▶ if a client gets a reference to the position or velocity of the projectile, then the client can change these quantities *without asking the projectile*
- ▶ this is not a flaw of aggregation
 - ▶ it's just the consequence of choosing to use aggregation

Composition

Composition

- ▶ recall that an object of type **X** that is composed of an object of type **Y** means
 - ▶ **X** has-a **Y** object *and*
 - ▶ **X** owns the **Y** object
- ▶ in other words

the **X** object has exclusive access to its **Y** object

Composition

the **X** object has exclusive access to its **Y** object

- ▶ this means that the **X** object will generally not share references to its **Y** object with clients
 - ▶ constructors will create new **Y** objects
 - ▶ accessors will return references to new **Y** objects
 - ▶ mutators will store references to new **Y** objects
- ▶ the “new **Y** objects” are called *defensive copies*

Composition & the Default Constructor

the X object has exclusive access to its Y object

- ▶ if a default constructor is defined it must create a suitable Y object

```
public X()  
{  
    // create a suitable Y; for example  
    this.y = new Y( /* suitable arguments */ );  
}
```

defensive copy

Composition & Other Constructors

the X object has exclusive access to its Y object

- ▶ a constructor that has a Y parameter must first deep copy and then validate the Y object

```
public X(Y y)
{
    // create a copy of y
    Y copyY = new Y(y);    } } defensive copy
    // validate; will throw an exception if copyY is invalid
    this.checkY(copyY);
    this.y = copyY;
}
```

Composition and Other Constructors

- ▶ why is the deep copy required?

the **X** object has exclusive access to its **Y** object

- ▶ if the constructor does this

```
// don't do this for composition
public X(Y y) {
    this.y = y;
}
```

then the client and the **X** object will share the same **Y** object

- ▶ this is called a privacy leak

Composition & Copy Constructor

the X object has exclusive access to its Y object

- ▶ if a copy constructor is defined it must create a new Y that is a deep copy of the other X object's Y object

```
public X(X other)
{
    // create a new Y that is a copy of other.y
    this.y = new Y(other.getY());
}
```

defensive copy

Composition & Copy Constructor

- ▶ what happens if the **X** copy constructor does not make a deep copy of the other **X** object's **Y** object?

```
// don't do this
public X(X other)
{
    this.y = other.y;
}
```

- ▶ every **X** object created with the copy constructor ends up sharing its **Y** object
 - ▶ if one **X** modifies its **Y** object, all **X** objects will end up with a modified **Y** object
 - ▶ this is called a privacy leak

Modify the **Projectile** copy constructor so that it uses composition:

```
/**  
 * Initialize the projectile so that its position and velocity are  
 * equal to the position and velocity of the specified projectile.  
 *  
 * @param other  
 *         a projectile to copy  
 */  
public Projectile(Projectile other) {  
    this.position =  
    this.velocity =  
}
```

Composition and Accessors

the X object has exclusive access to its Y object

- ▶ never return a reference to a field; always return a deep copy

```
public Y getY()  
{  
    return new Y(this.y); } } defensive copy
```

Composition and Accessors

- ▶ why is the deep copy required?

the **X** object has exclusive access to its **Y** object

- ▶ if the accessor does this

```
// don't do this for composition
public Y getY() {
    return this.y;
}
```

then the client and the **X** object will share the same **Y** object

- ▶ this is called a privacy leak

Modify the **Ball** accessor methods so that they use composition:

```
/*
 * Return the position of the projectile.
 *
 * @return the position of the projectile
 */
public Point2 getPosition() {
    return
}

/*
 * Return the velocity of the projectile.
 *
 * @return the velocity of the projectile
 */
public Vector2 getVelocity() {
    return
}
```

Composition and Mutators

the **X** object has exclusive access to its **Y** object

- ▶ if **X** has a method that sets its **Y** object to a client-provided **Y** object then the method must make a deep copy of the client-provided **Y** object and validate it

```
public void setY(Y y)
{
    Y copyY = new Y(y);    } } defensive copy
    // validate; will throw an exception if copyY is invalid
    this.checkY(copyY);
    this.y = copyY;
}
```

Composition and Mutators

- ▶ why is the deep copy required?

the **X** object has exclusive access to its **Y** object

- ▶ if the mutator does this

```
// don't do this for composition
public void setY(Y y) {
    this.y = y;
}
```

then the client and the **X** object will share the same **Y** object

- ▶ this is called a privacy leak

Modify the **Projectile** mutator methods so that they use composition:

```
/**  
 * Set the position of the projectile to the given position.  
 *  
 * @param position  
 *         the new position of the projectile  
 */  
public void setPosition(Point2 position) {  
    this.position =  
}  
  
/**  
 * Set the velocity of the projectile to the given velocity.  
 *  
 * @param velocity  
 *         the new velocity of the projectile  
 */  
public void setVelocity(Vector2 velocity) {  
    this.velocity =  
}
```

Price of Defensive Copying

- ▶ defensive copies are required when using composition, but the price of defensive copying is time and memory needed to create and garbage collect defensive copies of objects
- ▶ the **BouncingBall** program calls **move** followed by **getPosition** every 25 milliseconds
 - ▶ if **Projectile** uses composition then approximately 40 new **Point2** objects are created every second just to move one projectile

Composition (Part 2)

Class Invariants

- ▶ class invariant
 - ▶ some property of the state of the object that is established by a constructor and maintained between calls to public methods
 - ▶ in other words:
 - ▶ the constructor ensures that the class invariant holds when the constructor is finished running
 - the invariant does not necessarily hold while the constructor is running
 - ▶ every public method ensures that the class invariant holds when the method is finished running
 - the invariant does not necessarily hold while the method is running

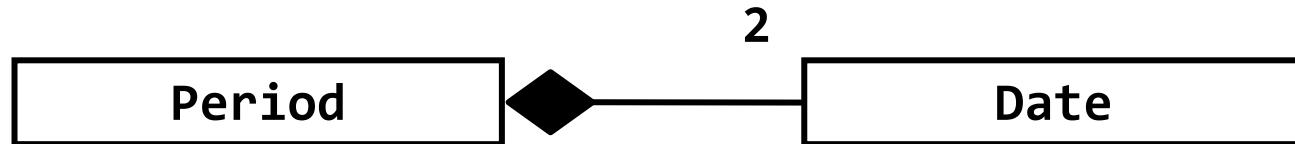
Period Class

- ▶ adapted from Effective Java by Joshua Bloch
 - ▶ available online at
<http://www.informit.com/articles/article.aspx?p=31551&seqNum=2>
- ▶ we want to implement a class that represents a period of time
 - ▶ a period has a start time and an end time
 - ▶ end time is always after the start time (this is the class invariant)

Period Class

- ▶ we want to implement a class that represents a period of time
 - ▶ has-a **Date** representing the start of the time period
 - ▶ has-a **Date** representing the end of the time period
 - ▶ class invariant: start of time period is always prior to the end of the time period

Period Class



Period is a composition
of two Date objects

java.util.Date

- ▶ <https://docs.oracle.com/javase/8/docs/api/java/util/Date.html>
- ▶ **Date** has no copy constructor; to copy a **Date** object do the following:

```
Date d = new Date();
Date d2 = new Date(d.getTime());
```

```
import java.util.Date;  
  
public class Period {  
    private Date start;  
    private Date end;
```

Suppose that we implement the **Period** constructor like so:

```
/**  
 * Initialize the period to the given start and end dates.  
 *  
 * @param start beginning of the period  
 * @param end end of the period; must not precede start  
 * @throws IllegalArgumentException if start is after end  
 */  
  
public Period(Date start, Date end) {  
    if (start.compareTo(end) > 0) {  
        throw new IllegalArgumentException("start after end");  
    }  
    this.start = start;  
    this.end = end;  
}
```

Add one more line of code to show how the client can break
the class invariant of **Period**:

```
Date start = new Date();
Date end = new Date( start.getTime() + 10000 );
Period p = new Period( start, end );
```

- A. `end.setTime(end.getTime());`
- B. `end.setTime(end.getTime() + 10000);`
- C. `start.setTime(end.getTime() - 1);`
- D. `end.setTime(start.getTime());`

Modify the **Period** constructor so that it uses composition:

```
/**  
 * Initialize the period to the given start and end dates.  
 *  
 * @param start beginning of the period  
 * @param end end of the period; must not precede start  
 * @throws IllegalArgumentException if start is after end  
 */  
  
public Period(Date start, Date end) {  
    Date startCopy = new Date(start.getTime());  
    Date endCopy = new Date(end.getTime());  
    if (startCopy.compareTo(endCopy) > 0) {  
        throw new IllegalArgumentException("start after end");  
    }  
    this.start = startCopy;  
    this.end = endCopy;  
}
```

Suppose that we implement the **Period** copy constructor like so:

```
/**  
 * Initialize the period so that it has the same start and end times  
 * as the specified period.  
 *  
 * @param other the period to copy  
 */  
public Period(Period other) {  
    this.start = other.start;  
    this.end = other.end;  
}
```

Modify the **Period** copy constructor so that it uses composition:

```
/**  
 * Initialize the period so that it has the same start and end times  
 * as the specified period.  
 *  
 * @param other the period to copy  
 */  
public Period(Period other) {  
    this.start = new Date(other.start.getTime());  
    this.end = new Date(other.end.getTime());  
}
```

Suppose that we implement the **Period** accessors like so:

```
/**  
 * Returns the starting date of the period.  
 *  
 * @return the starting date of the period  
 */  
public Date getStart() {  
    return this.start; — privacy leak  
}  
  
/**  
 * Returns the ending date of the period.  
 *  
 * @return the ending date of the period  
 */  
public Date getEnd() {  
    return this.end; — privacy leak  
}
```

Modify the **Period** accessors so that they use composition:

```
/**  
 * Returns the starting date of the period.  
 *  
 * @return the starting date of the period  
 */  
public Date getStart() {  
    return new Date(this.start.getTime());  
}  
  
/**  
 * Returns the ending date of the period.  
 *  
 * @return the ending date of the period  
 */  
public Date getEnd() {  
    return new Date(this.end.getTime());  
}
```

Suppose that we implement the **Period** mutator like so:

```
/*
 * Sets the starting date of the period.
 *
 * @param newStart the new starting date of the period
 * @return true if the new starting date is earlier than the
 *         current end date; false otherwise
 */
public boolean setStart(Date newStart) {
    boolean ok = false;
    if (newStart.compareTo(this.end) < 0) {
        this.start = newStart;
        ok = true;
    }
    return ok;
}
```

Modify the **Period** mutator so that it uses composition:

```
/*
 * Sets the starting date of the period.
 *
 * @param newStart the new starting date of the period
 * @return true if the new starting date is earlier than the
 *         current end date; false otherwise
 */
public boolean setStart(Date newStart) {
    boolean ok = false;
    Date copy = new Date(newStart.getTime());
    if (copy.compareTo(this.end) < 0) {
        this.start = copy;
        ok = true;
    }
    return ok;
}
```

Privacy Leaks

- ▶ a privacy leak occurs when a class exposes a reference to a non-public field (that is not a primitive or immutable)
- ▶ given a class X that is a composition of a Y

```
public class X {  
    private Y y;  
    // ...  
}
```

these are all examples of privacy leaks

```
public X(Y y) {  
    this.y = y;  
}
```

```
public X(X other) {  
    this.y = other.y;  
}
```

```
public Y getY() {  
    return this.y;  
}
```

```
public void setY(Y y) {  
    this.y = y;  
}
```

Consequences of Privacy Leaks

- ▶ a privacy leak allows some other object to control the state of the object that leaked the field
- ▶ the object state can become inconsistent
 - ▶ example: if a **CreditCard** exposes a reference to its expiry **Date** then a client could set the expiry date to before the issue date

Consequences of Privacy Leaks

- ▶ a privacy leak allows some other object to control the state of the object that leaked the field
- ▶ it becomes impossible to guarantee class invariants
 - ▶ example: if a **Period** exposes a reference to one of its **Date** objects then the end of the period could be set to before the start of the period

Consequences of Privacy Leaks

- ▶ a privacy leak allows some other object to control the state of the object that leaked the field
- ▶ composition becomes broken because the object no longer owns its attribute
 - ▶ when an object “dies” its parts may not die with it

Recipe for Immutability

- ▶ the recipe for immutability in Java is described by Joshua Bloch in the book *Effective Java**
1. Do not provide any methods that can alter the state of the object
 2. Prevent the class from being extended
 3. Make all fields **final**
 4. Make all fields **private**
 5. Prevent clients from obtaining a reference to any mutable fields

revisit when we talk about inheritance

Composition with an array

- ▶ in Java an array is a reference type
 - ▶ if a class has a field that is an array, then you need to consider whether you should be using aggregation or composition
- ▶ an array of elements having primitive type is easier to deal with than an array of elements having reference type
 - ▶ if your class has a field that is an array of elements having reference type then things become more complicated
 - ▶ more on this next lecture

CombinationLock

- ▶ suppose that you want to implement a simple combination lock
 - ▶ the lock has a combination having at least 3 digits between 0 and 9
 - ▶ the lock is always either locked or unlocked
-
- ▶ fields:
 - ▶ an array of int to store the combination
 - ▶ a boolean to store the locked/unlocked state

```
import java.util.Arrays;

public class CombinationLock implements Comparable<CombinationLock> {

    /**
     * The combination of this lock.
     */
    private int[] combination;

    /**
     * The state of the lock.
     */
    private boolean isLocked;
```

CombinationLock

- ▶ note that the field **this.combination** is not yet initialized
- ▶ the constructors must make a new array and store reference to the new array in **this.combination**

```
/**  
 * Initializes this lock so that it is locked and it has the  
 * combination {@code 9, 9, 9}.  
 */  
public CombinationLock() {  
    this.combination = new int[3];  
    for (int i = 0; i < 3; i++) {  
        this.combination[i] = 9;  
    }  
    this.isLocked = true;  
}
```

CombinationLock

- ▶ a constructor that receives a combination from the caller must make a new copy of the combination and then validate the copy
- ▶ failing to at least make a copy of the combination is a potential privacy leak

```
/**  
 * Initializes this lock so that it is locked and it has the specified  
 * combination. The constructor copies the values from the specified  
 * combination into this lock's combination.  
 *  
 * @param combination  
 *         the combination to use for this lock  
 * @pre. combination.length greater than or equal to 3  
 * @throws IllegalArgumentException  
 *         if the number of numbers in the combination is less than 3  
 */  
  
public CombinationLock(int[] combination) {  
    int[] comboCopy = Arrays.copyOf(combination, combination.length);  
    int n = comboCopy.length;  
    if (n < 3) {  
        throw new IllegalArgumentException();  
    }  
    this.combination = comboCopy;  
    this.isLocked = true;  
}
```

CombinationLock

- ▶ the copy constructor must make a copy of the other lock's combination
- ▶ the easiest way to do this is to use constructor chaining at the price of performing some unnecessary validation

```
/**  
 * Initializes this lock by copying the digits from another combination  
 * lock into this lock's combination. Also copies the lock state of  
 * the other lock (if the other lock is locked then so is this lock,  
 * and if other lock is unlocked then so is this lock).  
 *  
 * @param other the lock to copy  
 */  
public CombinationLock(CombinationLock other) {  
    this(other.combination);  
    this.isLocked = other.isLocked;  
}
```

CombinationLock

- ▶ not every method needs to perform a defensive copy
 - ▶ unlocking the lock requires the caller to provide a combination with which to try to unlock the lock with
 - ▶ the provided combination is not used to modify the lock's combination and the method does not return the the lock's combination so there is no privacy leak here

```
/**  
 * Unlocks this lock if the provided combination is equal to the combination  
 * of this lock. If the provided combination does not match the combination  
 * of this lock then no action is taken.  
 *  
 * <p>  
 * If the specified combination is equal to the combination of this lock then  
 * {@code isLocked()} will return {@code false} after this method is called.  
 *  
 * @param combination  
 *         a combination to try to unlock this lock  
 */  
  
public void unlock(int[] combination) {  
    if (Arrays.equals(this.combination, combination)) {  
        this.isLocked = false;  
    }  
}
```

Collections as Fields

Still Aggregation and Composition

Motivation

- ▶ often you will want to implement a class that has-a collection as a field
 - ▶ a university has-a collection of faculties and each faculty has-a collection of schools and departments
 - ▶ a receipt has-a collection of items
 - ▶ a contact list has-a collection of contacts
 - ▶ from the notes, a student has-a collection of GPAs and has-a collection of courses
 - ▶ a polygonal model has-a collection of triangles*

*polygons, actually, but triangles are easier to work with

What Does a Collection Hold?

- ▶ a collection holds references to instances
 - ▶ it does not hold the instances

```
ArrayList<Date> dates =  
    new ArrayList<Date>();
```

```
Date d1 = new Date();  
Date d2 = new Date();  
Date d3 = new Date();
```

```
dates.add(d1);  
dates.add(d2);  
dates.add(d3);
```

100	client invocation
dates	200a
d1	500a
d2	600a
d3	700a
...	
200	ArrayList object
	500a
	600a
	700a

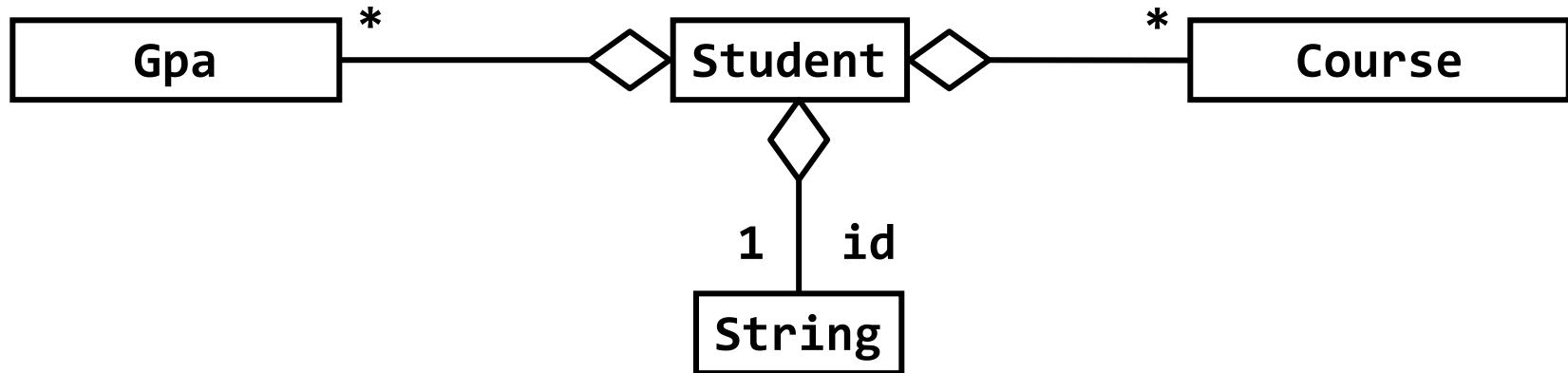
► What does the following print?

```
ArrayList<Point2> pts = new ArrayList<Point2>();  
Point2 p = new Point2(0., 0.);  
pts.add(p);  
p.x( 10.0 );  
System.out.print(p);  
System.out.println(", " + pts.get(0));
```

- A. (0.0, 0.0), (0.0, 0.0)
- B. (0.0, 0.0), (10.0, 0.0)
- C. (10.0, 0.0), (0.0, 0.0)
- D. (10.0, 0.0), (10.0, 0.0)

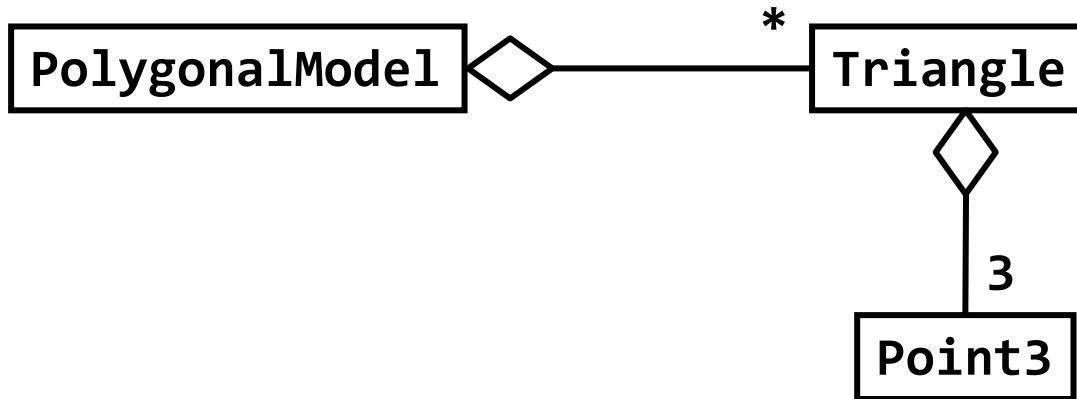
Student Class

- ▶ a **Student** has-a string id
- ▶ a **Student** has-a collection of yearly GPAs
- ▶ a **Student** has-a collection of courses

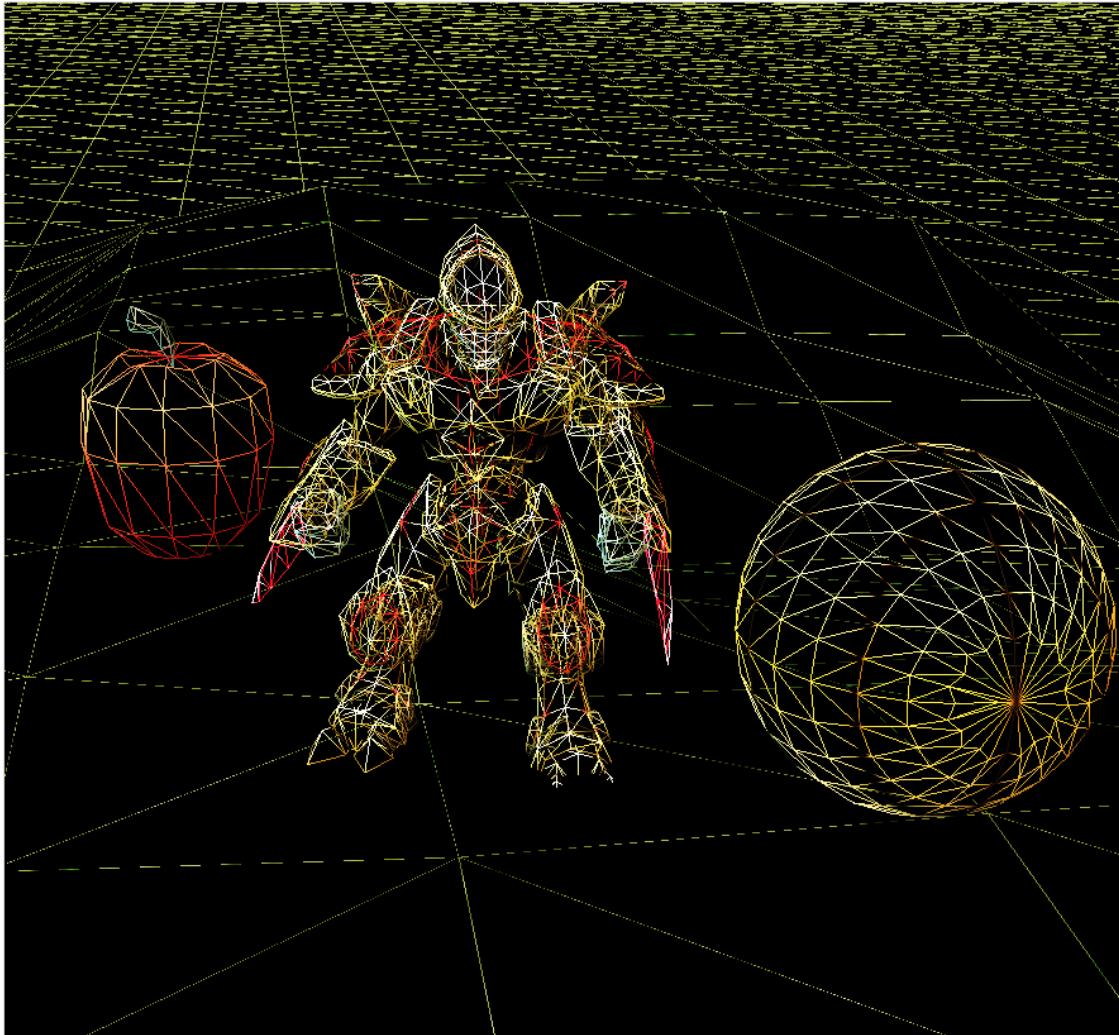


PolygonalModel Class

- ▶ a polygonal model has-a collection of **Triangles**
- ▶ aggregation







PolygonalModel

```
class PolygonalModel {  
  
    private List<Triangle> tri;  
  
    public PolygonalModel() {  
        this.tri = new ArrayList<Triangle>();  
    }  
  
}
```

PolygonalModel

```
public void clear() {  
    // removes all Triangles  
    this.tri.clear();  
}  
  
public int size() {  
    // returns the number of Triangles  
    return this.tri.size();  
}
```

Collections as Fields

- ▶ when using a collection as an field of a class **X** you need to decide on ownership issues
 - ▶ does **X** own or share its collection?
 - ▶ if **X** owns the collection, does **X** own the objects held in the collection?

X Shares its Collection with other Xs

- ▶ if X shares its collection with other X instances, then the copy constructor does not need to create a new collection
 - ▶ the copy constructor can simply assign its collection
 - ▶ X's collection is an alias for another collection

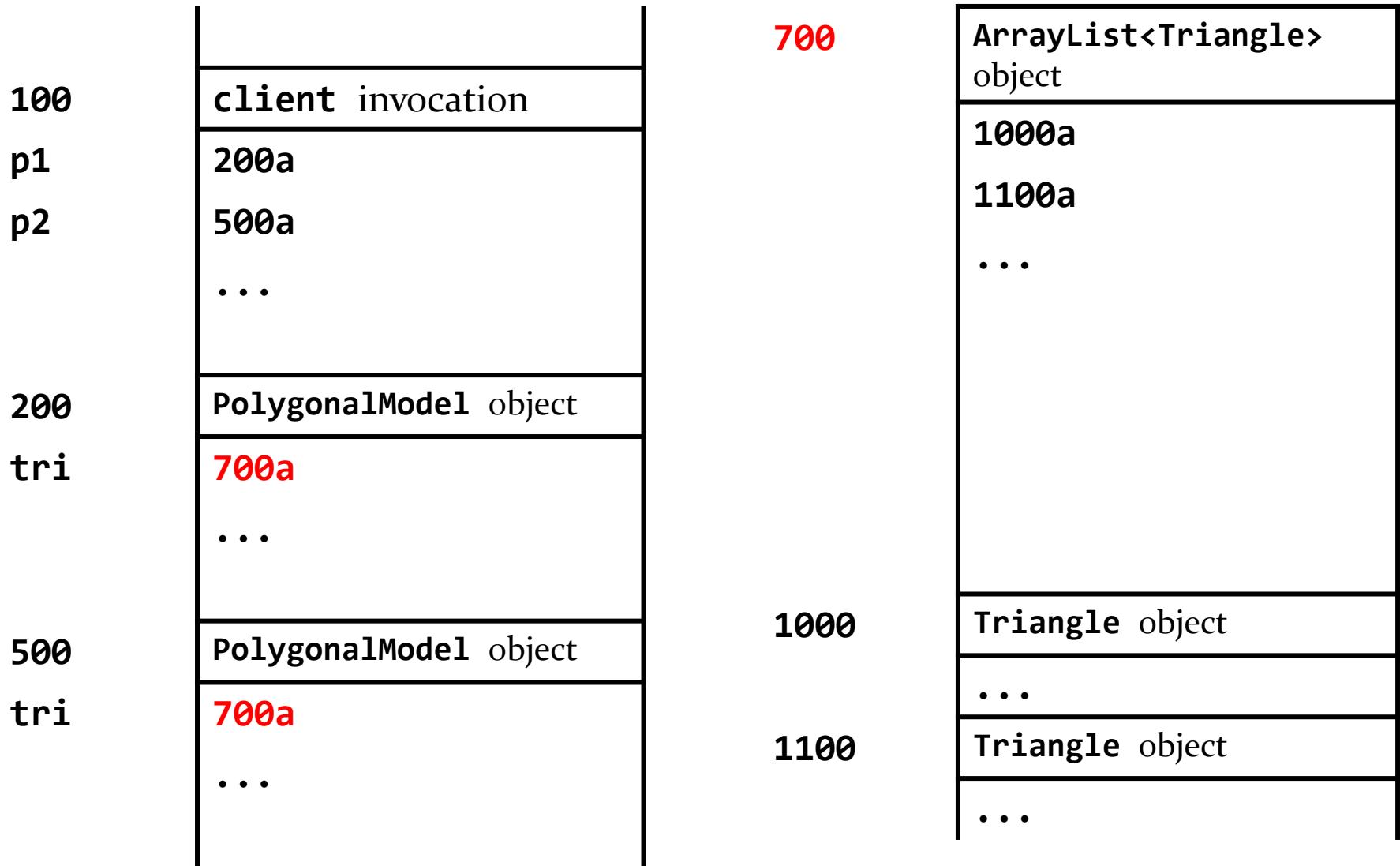
PolygonalModel Copy Constructor 1

```
public PolygonalModel(PolygonalModel other) {  
    // implements aliasing (sharing) with other  
    // PolygonalModel instances  
    this.tri = other.tri;  
}  
  
public List<Triangle> getTriangles() {  
    return this.tri;  
}
```

alias: no new List created

alias: no new List created

```
PolygonalModel p2 = new PolygonalModel(p1);
```



-
- ▶ Suppose that the **PolygonalModel** copy constructor makes an alias of the list of triangles.
Suppose you have a **PolygonalModel** **p1** that has 100 **Triangles**. What does the following code print?

```
PolygonalModel p2 = new PolygonalModel(p1);
p2.clear();
System.out.print( p2.size() );
System.out.println( ", " + p1.size() );
```

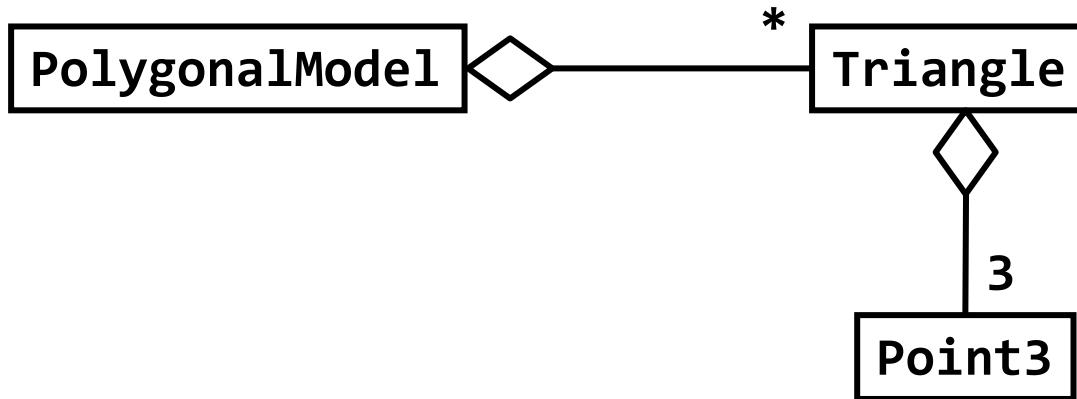
- A. 0, 0
- B. 0, 100

X Owns its Collection: Shallow Copy

- ▶ if X owns its collection but not the objects in the collection then the copy constructor can perform a shallow copy of the collection
- ▶ a shallow copy of a collection means
 - ▶ X creates a new collection, but it does not make new copies of the elements in the collection
 - ▶ the references in the collection are aliases for references in the other collection

PolygonalModel Class 2

- ▶ a polygonal model has-a collection of **Triangles**
 - ▶ still an aggregation of triangles



X Owns its Collection: Shallow Copy

- ▶ the hard way to perform a shallow copy of a list named **dates**

shallow copy: new **List**
created but elements
are all aliases

```
ArrayList<Date> sCopy = new ArrayList<Date>();  
for(Date d : dates) {  
    sCopy.add(d);  
}
```

add adds an alias of **d**
to **sCopy**

X Owns its Collection: Shallow Copy

- ▶ the easy way to perform a shallow copy of a list named **dates**

```
ArrayList<Date> sCopy = new ArrayList<Date>(dates);
```

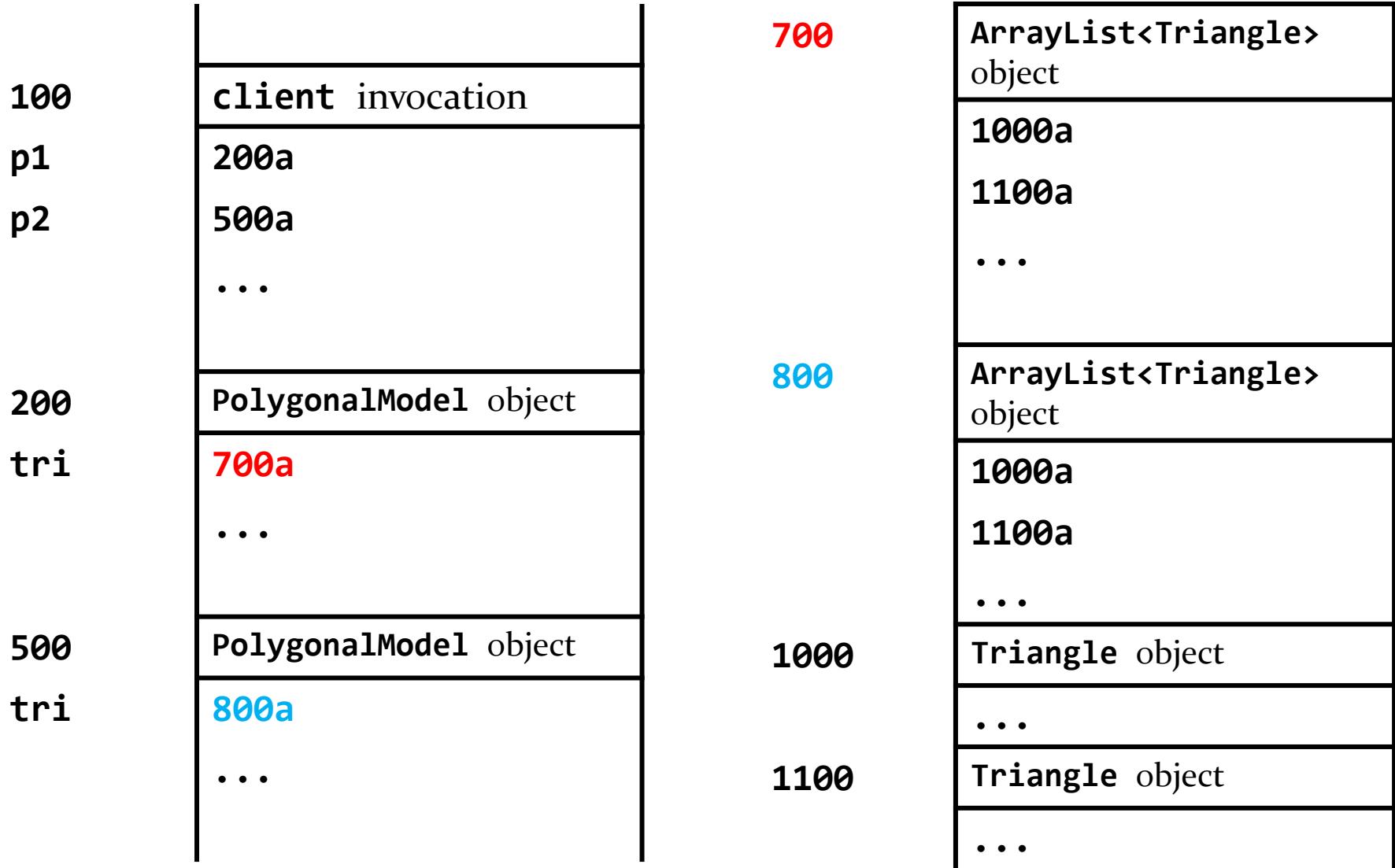
- ▶ **List** and **Set** constructors that have a **Collection** as a parameter make a shallow copy of the **Collection**

PolygonalModel Copy Constructor 2

```
public PolygonalModel(PolygonalModel other) {  
    // implements shallow copying  
    this.tri = new ArrayList<Triangle>(other.tri);  
}
```

shallow copy: new **List**
created, but no new
Triangle objects created

```
PolygonalModel p2 = new PolygonalModel(p1);
```



-
- ▶ Suppose that the **PolygonalModel** copy constructor makes a shallow copy of the list of triangles.
Suppose you have a **PolygonalModel** **p1** that has 100 **Triangles**. What does the following code print?

```
PolygonalModel p2 = new PolygonalModel(p1);
p2.clear();
System.out.print( p2.size() );
System.out.println( ", " + p1.size() );
```

- A. 0, 0
- B. 0, 100

-
- ▶ Suppose that the **PolygonalModel** copy constructor makes a shallow copy of the list of triangles.
Suppose you have a **PolygonalModel** **p1** that has 100 **Triangles**. What does the following code print?

```
PolygonalModel p2 = new PolygonalModel(p1);
Triangle t1 = p1.getTriangles().get(0);
Triangle t2 = p2.getTriangles().get(0);
System.out.println(t1 == t2);
```

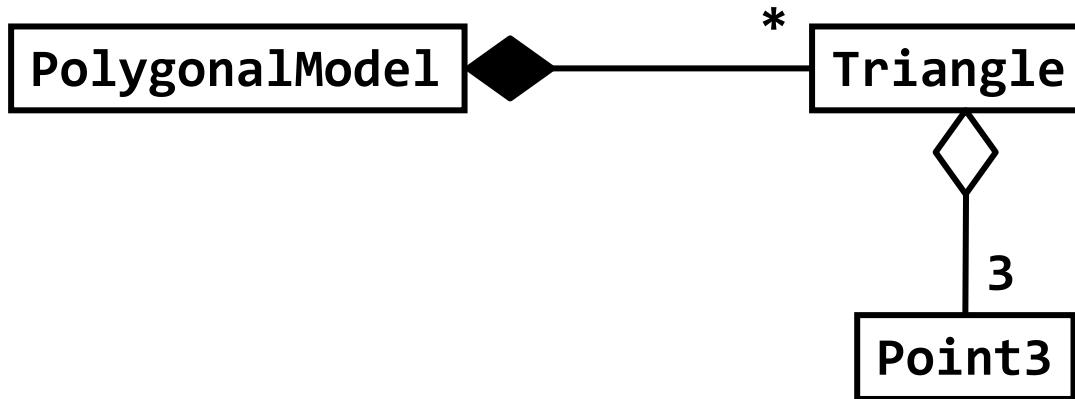
- A. **false**
- B. **true**

X Owns its Collection: Deep Copy

- ▶ if X owns its collection and the objects in the collection then the copy constructor must perform a deep copy of the collection
- ▶ a deep copy of a collection means
 - ▶ X creates a new collection, and it creates deep copies of all of the elements of the collection
 - ▶ the references in the collection are references to new objects (that are copies of the objects in other collection)

PolygonalModel Class 2

- ▶ a polygonal model has-a collection of **Triangles**
- ▶ composition of triangles



X Owns its Collection: Deep Copy

- ▶ how to perform a deep copy of a list named **dates**

```
ArrayList<Date> dCopy = new ArrayList<Date>();  
for(Date d : dates) {  
    dCopy.add(new Date(d.getTime()));  
}  
  
new Date created that  
is a copy of d
```

deep copy: new List
created and new
elements created

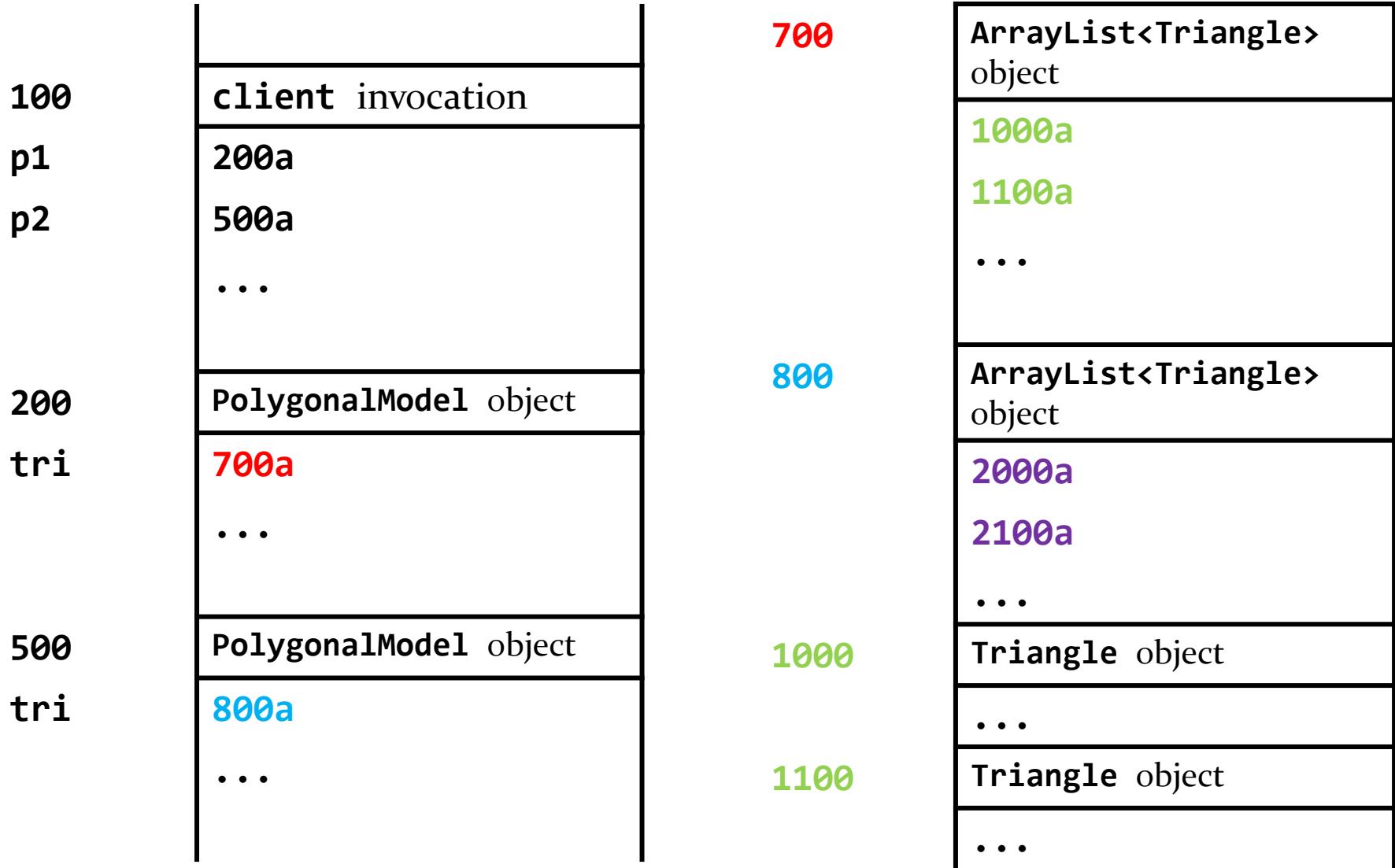
PolygonalModel Copy Constructor 3

```
public PolygonalModel(PolygonalModel other) {  
  
    // implements deep copying  
    this.tri = new ArrayList<Triangle>();  
    for (Triangle t : other.getTriangles()) {  
        this.tri.add(new Triangle(t));  
    }  
}
```

deep copy: new **List** created, and new **Triangle** objects created

new **Triangle** created that is a copy of **t**

```
PolygonalModel p2 = new PolygonalModel(p1);
```



2000

Triangle object

...

2100

Triangle object

...

-
- ▶ Suppose that the **PolygonalModel** copy constructor makes a deep copy of the list of triangles.
Suppose you have a **PolygonalModel** **p1** that has 100 **Triangles**. What does the following code print?

```
PolygonalModel p2 = new PolygonalModel(p1);
p2.clear();
System.out.println( p2.size() );
System.out.println( p1.size() );
```

- A. 0, 0
- B. 0, 100

-
- ▶ Suppose that the **PolygonalModel** copy constructor makes a deep copy of the list of triangles.
Suppose you have a **PolygonalModel** **p1** that has 100 **Triangles**. What does the following code print?

```
PolygonalModel p2 = new PolygonalModel(p1);
Triangle t1 = p1.getTriangles().get(0);
Triangle t2 = p2.getTriangles().get(0);
System.out.print( t1 == t2 );
System.out.println( ", " + t1.equals(t2) );
```

- A. **false, true**
- B. **true, true**

Interfaces

Interfaces

- ▶ recall that an interface is a specification of the methods that an implementing class must provide
- ▶ usually made up of one or more method declarations without any method implementations
 - ▶ static method implementations are allowed
 - ▶ default method implementation are allowed

Interfaces

- ▶ recall that an interface defines a type
 - ▶ can create variables of interface type, e.g.,

```
List<String> t = new ArrayList<>();  
t = new LinkedList<>();  
t = new MyList<>();  
// previous line ok only if  
// MyList implements List
```

Implementing interfaces

- ▶ recall that a class that implements an interface must implement all of the methods declared by the interface
- ▶ implementing **Comparable** requires implementing **compareTo**

The Iterator interface

- ▶ an **Iterator** is an object that knows how to iterate over a collection of elements or a sequence of values
- ▶ using an **Iterator** object is the only guaranteed safe way of iterating over a **List** or **Set** and removing elements during the iteration
 - ▶ see **List** and **Set** notebooks for details
- ▶ Java's for-each loop actually uses an **Iterator** object
 - ▶ the language hides the object from the programmer by having the compiler create the object and call the appropriate methods

The Iterator interface

```
package java.util;

public interface Iterator<T> {

    public boolean hasNext();
    public T next();

    // two more default methods not shown here
}
```

The **Iterable** interface

- ▶ any object whose class implements the **Iterable** interface can be the target of a for-each loop
 - ▶ **List<E>** implements **Collection<E>**, **Iterable<E>**
 - ▶ **Set<E>** implements **Collection<E>**, **Iterable<E>**
 - ▶ arrays do not implement the **Iterable** interface
 - ▶ but the language pretends that they do

The Iterable interface

```
package java.lang;

public interface Iterable<T> {

    public Iterator<T> iterator();

    // two more default methods not shown here
}
```

Python loops

- ▶ in Python, a counting-style loop involves iterating over a **range** object:

```
for i in range(0, 10):  
    # i takes on values 0, 1, 2, ..., 9
```

Emulating Python loops

- ▶ we can emulate a Python counting-style loop by creating a **Range** class that implements **Iterable<Integer>**:

```
for (int i : new Range(0, 10)) {  
    # i takes on values 0, 1, 2, ..., 9  
}
```

```
import java.lang.Iterable;
import java.util.Iterator;
import java.util.NoSuchElementException;

public class Range implements Iterable<Integer> {
    private int start;
    private int stop;

    public Range(int stop) {
        this(0, stop);
    }

    public Range(int start, int stop) {
        if (stop < start) {
            throw new IllegalArgumentException("stop less than start");
        }
        this.start = start;
        this.stop = stop;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            private int current = start;

            @Override
            public boolean hasNext() {
                return current < stop;
            }

            @Override
            public Integer next() {
                if (!hasNext()) {
                    throw new NoSuchElementException();
                }
                return current++;
            }
        };
    }
}
```

Makes **Range** objects a target for a for-each loop.

```
@Override  
public Iterator<Integer> iterator() {  
    return new RangeIterator();  
}  
  
/**  
 * An iterator over a range. The iterator starts at the starting value  
 * of the range and goes up to, but not including, the maximum value of  
 * the range.  
 */  
  
private class RangeIterator implements Iterator<Integer> {  
}
```

Required by the **Iterable** interface.

A nested, inner class (or just inner class).

Only **Range** objects can make **RangeIterator** objects.

Nested class

- ▶ a nested class is a class that is defined inside of another class
- ▶ a nested class usually exists only to serve the class that it is defined in
 - ▶ if the nested class would be useful on its own, then it should be a top-level class
 - ▶ but see `Map.Entry` for a counter-example
- ▶ iterators are usually defined as nested class because you can't have an iterator without its enclosing class
 - ▶ e.g., a `ListIterator` without a `List` makes no sense
 - ▶ e.g., a `RangeIterator` without a `Range` makes no sense

Inner classes

- ▶ a nested class is an inner class if the inner class is *not* marked as being **static**
- ▶ objects of an inner class type always have a reference to an object of the enclosing class
 - ▶ e.g., a **RangeIterator** object has a reference to a **Range** object
 - ▶ an inner class object can access the fields of its enclosing class object
 - ▶ e.g., a **RangeIterator** object can access the fields of its enclosing **Range** object

What does a **RangeIterator** object do?

- ▶ a newly created **RangeIterator** object:
 - ▶ starts counting at the starting value of its range
 - ▶ every time **next()** is called:
 - ▶ throws an exception if the count has reached the stopping value of its range,
 - ▶ otherwise, returns the current count and adds one to the count
 - ▶ every time **hasNext()** is called:
 - ▶ returns **true** if the current count is less than the stopping value,
 - ▶ otherwise, returns **false**

```
private class RangeIterator implements Iterator<Integer> {  
    // the value returned by next  
    private int val;  
  
    // starts counting from the starting value of the  
    // enclosing Range object  
    RangeIterator() {  
        this.val = Range.this.start;  
    }  
  
    // returns true if this.val is less than the stopping  
    // value of the enclosing Range object  
    @Override  
    public boolean hasNext() {  
        return this.val < Range.this.stop;  
    }  
}
```

Somewhat unusual syntax
to access the fields of the
enclosing **Range** object

```
// returns the current value of this.val and
// then increments this.val
@Override
public Integer next() {
    if (!this.hasNext()) {
        throw new NoSuchElementException("no more
elements in range");
    }
    Integer result = this.val;
    this.val++;
    return result;
}

} // end RangeIterator

} // end Range
```

```
for (int i : new Range(0, 10)) {  
    System.out.println(i);  
}
```

outputs:

0
1
2
3
4
5
6
7
8
9

Creating an interface

- ▶ to create an interface, simply declare the interface similarly to how you would declare a class
 - ▶ but change **class** to **interface**
- ▶ everything in an interface is **public**
 - ▶ even if you do not include an access modifier
 - ▶ using any modifier other than **public** results in a compile-time error

Creating an interface

```
/**  
 * An interface for objects that can be printed  
 * to standard output. Not a particularly useful  
 * interface.  
 */  
public interface Printable {  
}
```

Creating an interface

- ▶ add the method declarations for your interface
 - ▶ each method declaration must end with a semi-colon

Creating an interface

```
/**  
 * An interface for objects that can be printed  
 * to standard output. Not a particularly useful  
 * interface.  
 */  
public interface Printable {  
    /**  
     * Prints this object to standard output.  
     */  
    public void print();  
}
```

Create an implementing class

- ▶ we can easily modify one of our existing classes to implement the **Printable** interface
- ▶ e.g., the **Point2** class

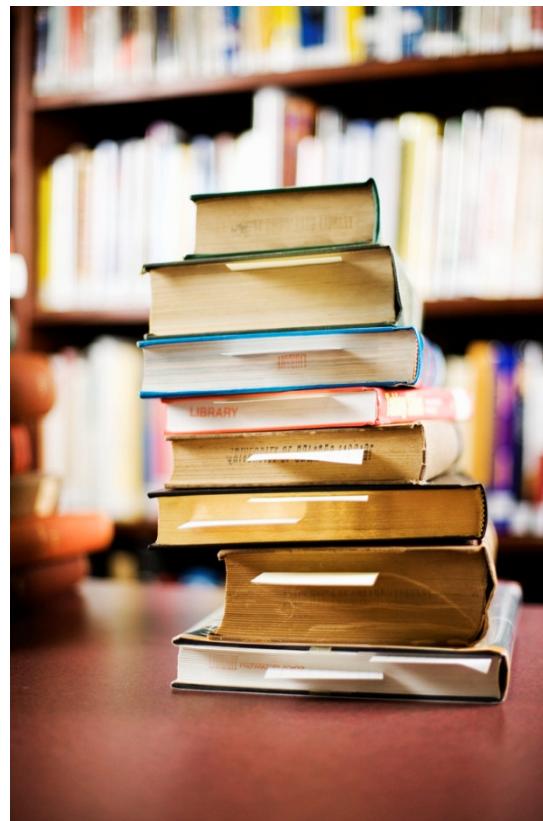
```
public class Point2
implements Comparable<Point2>, Printable {
    // rest of class same as before
    @Override
    public void print() {
        System.out.println(this.toString());
    }
} // end Point2
```

Classes can implement multiple interfaces.

Stacks

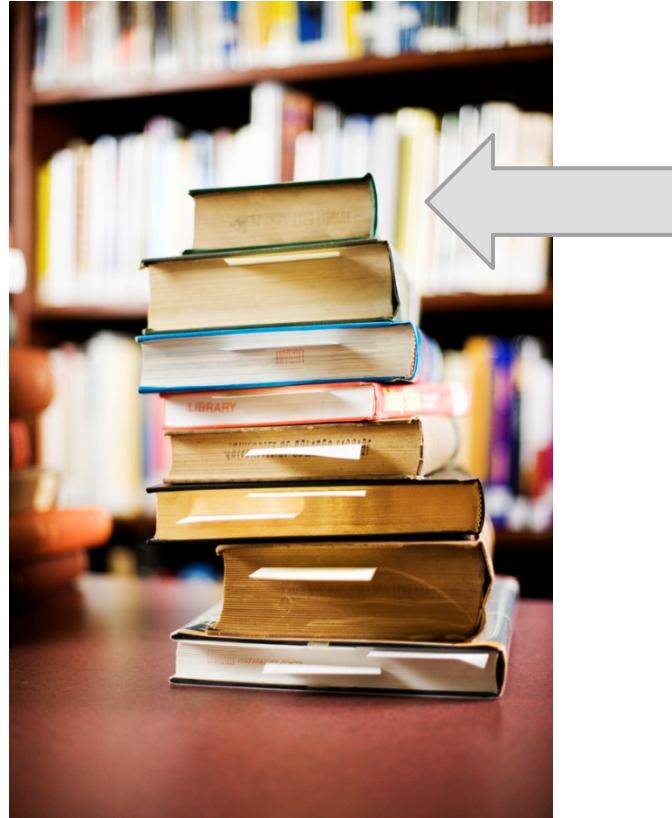
Stack

- examples of stacks



Top of Stack

- ▶ top of the stack

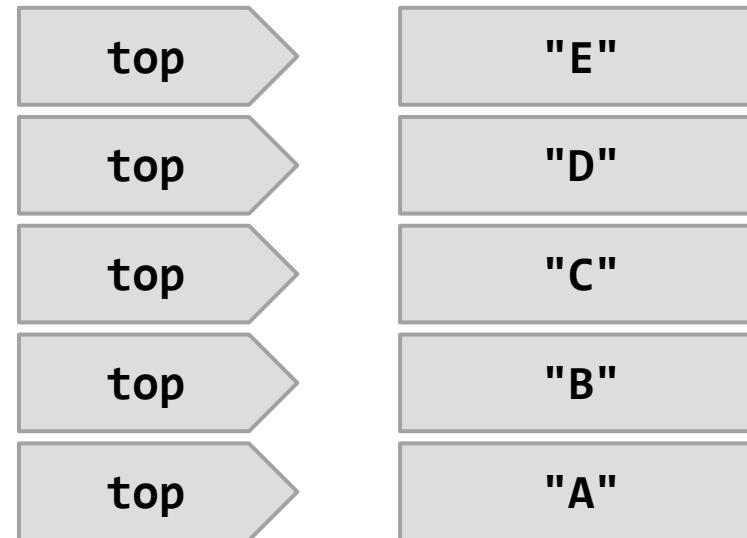


Stack Operations

- ▶ classically, stacks only support two operations
 - 1. push
 - ▶ add to the top of the stack
 - 2. pop
 - ▶ remove from the top of the stack
 - ▶ throws an exception if there is nothing on the stack

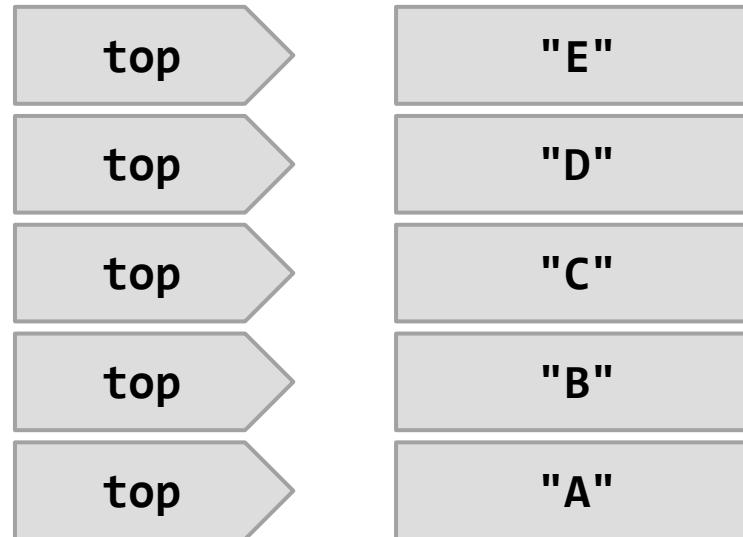
Push

1. **st.push("A")**
2. **st.push("B")**
3. **st.push("C")**
4. **st.push("D")**
5. **st.push("E")**



Pop

1. **String s = st.pop()**
2. **s = st.pop()**
3. **s = st.pop()**
4. **s = st.pop()**
5. **s = st.pop()**



Applications

- ▶ stacks are used widely in computer science and computer engineering
 - ▶ undo/redo
 - ▶ widely used in parsing
 - ▶ a call stack is used to store information about the active methods in a Java program
 - ▶ convert a recursive method into a non-recursive one

Example: Reversing a sequence

- ▶ a silly and usually inefficient way to reverse a sequence is to use a stack

Don't do this

```
public static List<String> reverse(List<String> t) {  
    List<String> result = new ArrayList<>();  
    Stack<String> st = new Stack<>();  
    for (String s : t) {  
        st.push(s);  
    }  
    while (!st.isEmpty()) {  
        result.add(st.pop());  
    }  
    return result;  
}
```

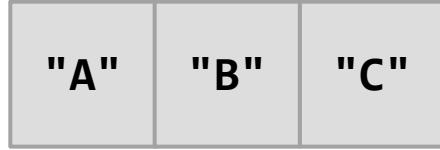
"A"	"B"	"C"
-----	-----	-----

list **t**

stack

list **result**





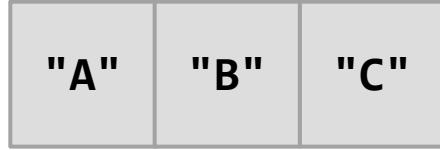
list **t**



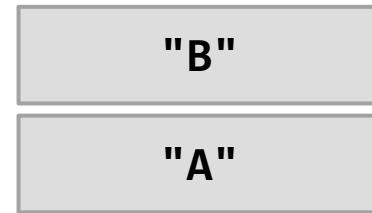
stack

list **result**





list **t**



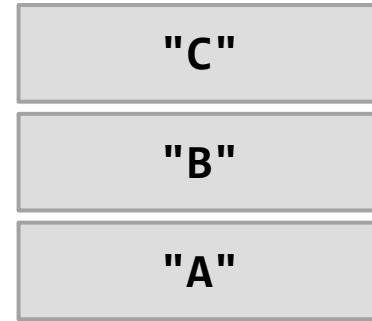
stack

list **result**





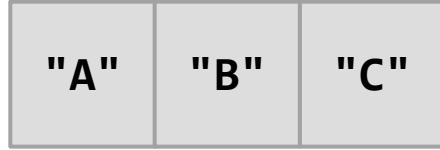
list **t**



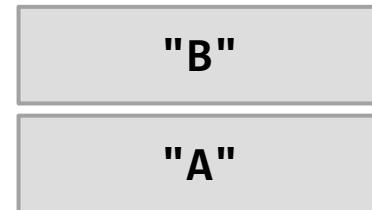
stack

list **result**





list **t**



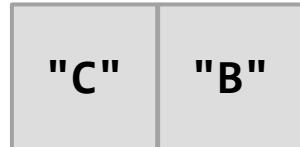
stack

list **result**





list **t**



stack

list **result**



"A"	"B"	"C"
-----	-----	-----

list **t**

"C"	"B"	"A"
-----	-----	-----

stack

list **result**



Implementing a stack

- ▶ a stack looks a lot like a list
 - ▶ pushing an element onto the top of the stack looks like adding an element to the end of a list
 - ▶ popping an element from the top of a stack looks like removing an element from the end of the list

```
import java.util.ArrayList;

public class Stack {

    private ArrayList<String> stack;

    public Stack() {
        this.stack = new ArrayList<>();
    }

    public int size() {
        return this.stack.size();
    }

    public void push(String elem) {
        this.stack.add(elem);
    }

    public String pop() {
        String elem = this.stack.remove(this.size() - 1);
        return elem;
    }
}
```

Add element to end of list.
Usually in $O(1)$.

Remove element at end of
list. In $O(1)$.

Alternative implementations

- ▶ there are at least two other ways of implementing a stack
 - ▶ use an array to store the elements
 - ▶ need to handle reallocating the array when the array becomes full
 - ▶ use a linked structure
- ▶ both alternative implementations have their own advantages and disadvantages
 - ▶ create an interface, implement the different versions and let the user pick which one is most appropriate for their purposes

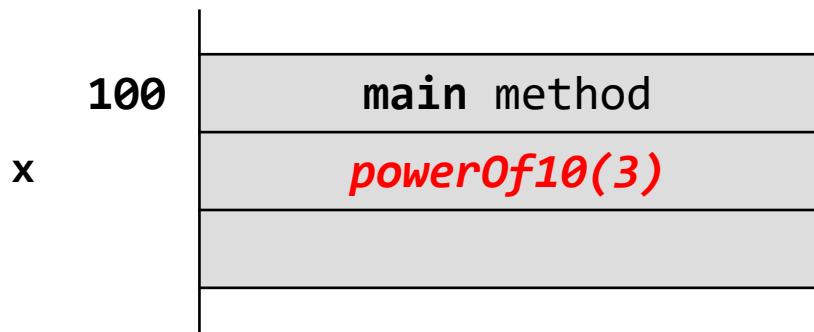
Stacks

Converting a Recursive Method

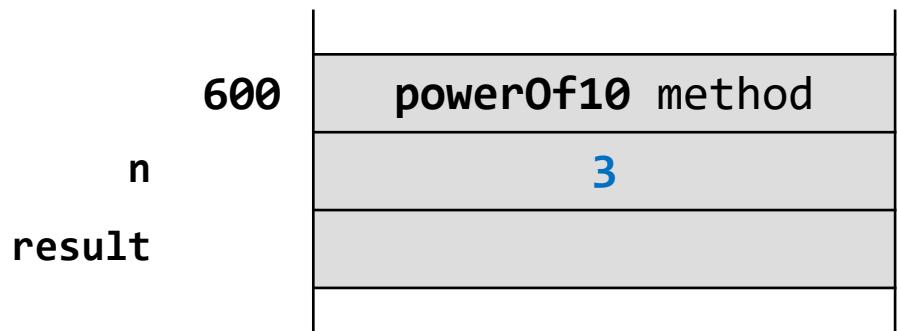
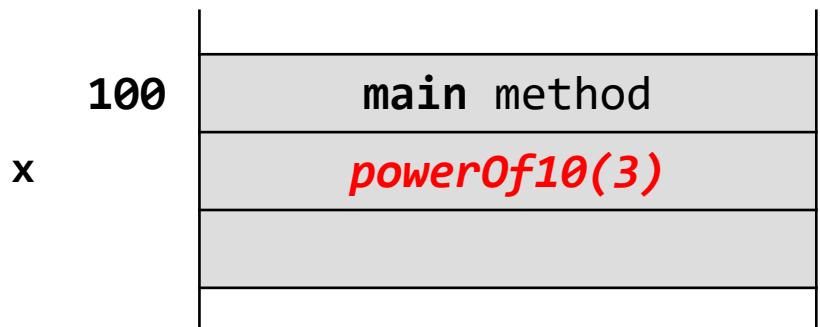
- ▶ a stack can be used to convert a recursive method to a non-recursive method
- ▶ the key to understanding how to do this lies in the memory diagram for a recursive memory

```
public static double powerOf10(int n) {  
    double result;  
    if (n == 0) {  
        result = 1.0;  
    }  
    else {  
        result = 10 * powerOf10(n - 1);  
    }  
    return result;  
}
```

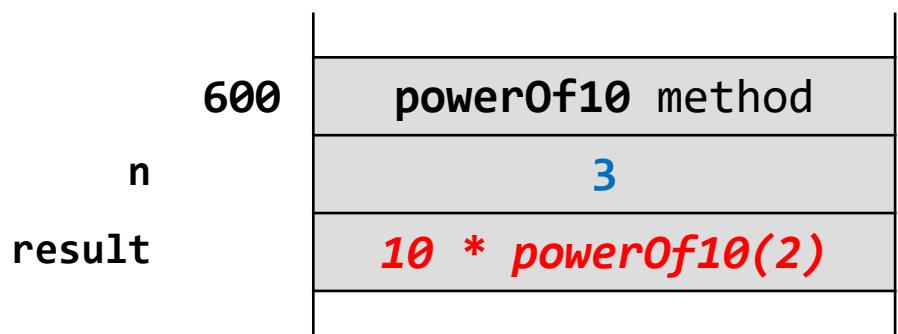
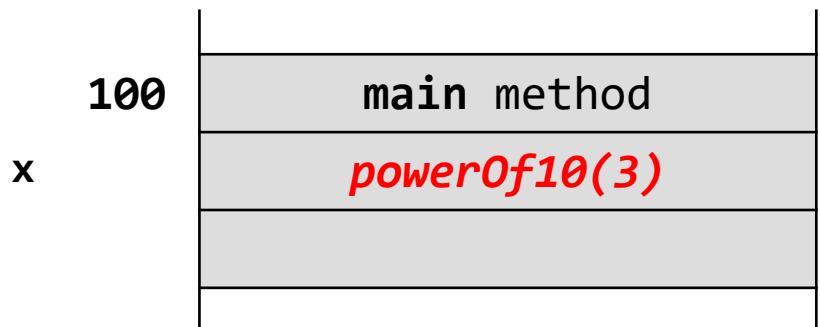
```
double x = Recursion.powerOf10(3);
```



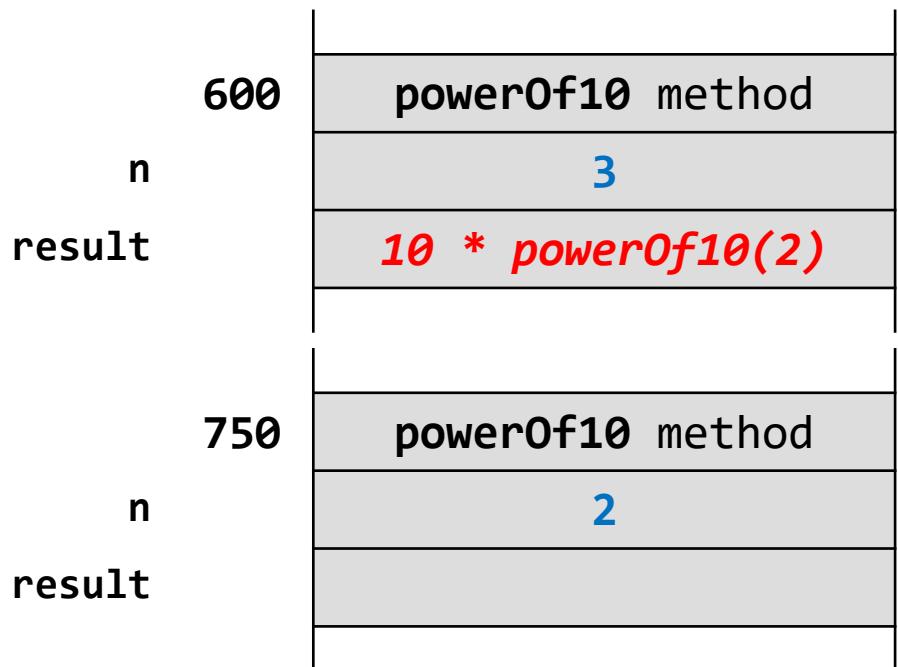
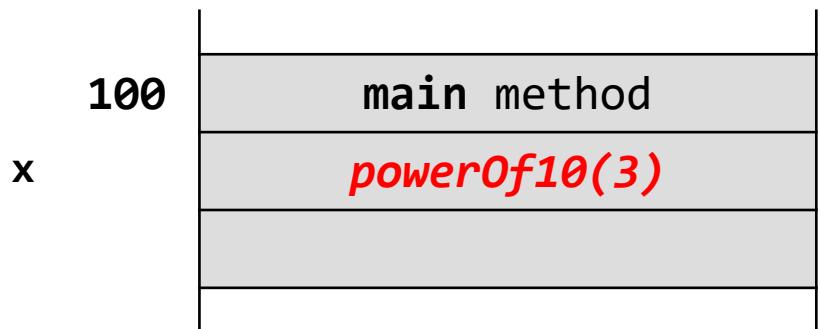
```
double x = Recursion.powerOf10(3);
```



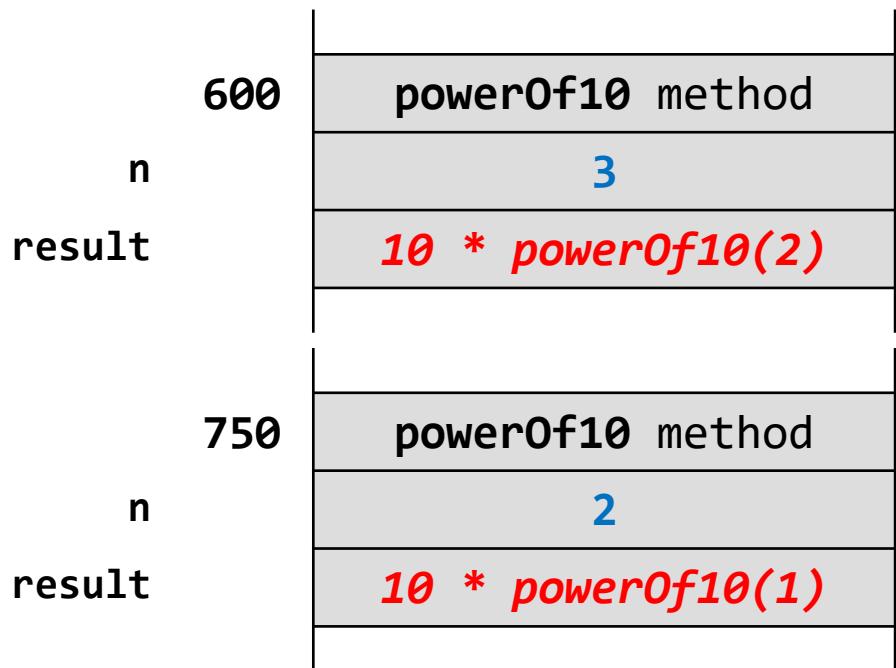
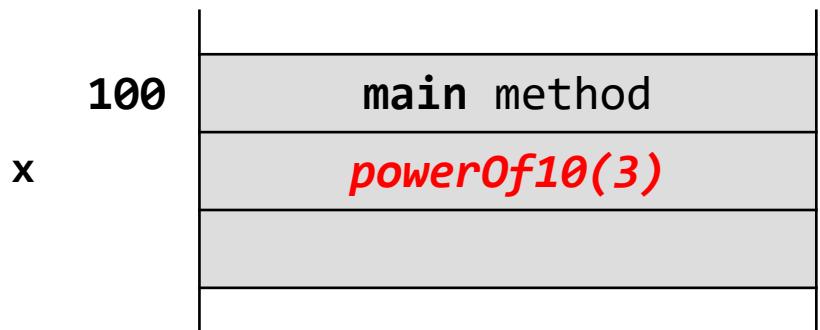
```
double x = Recursion.powerOf10(3);
```



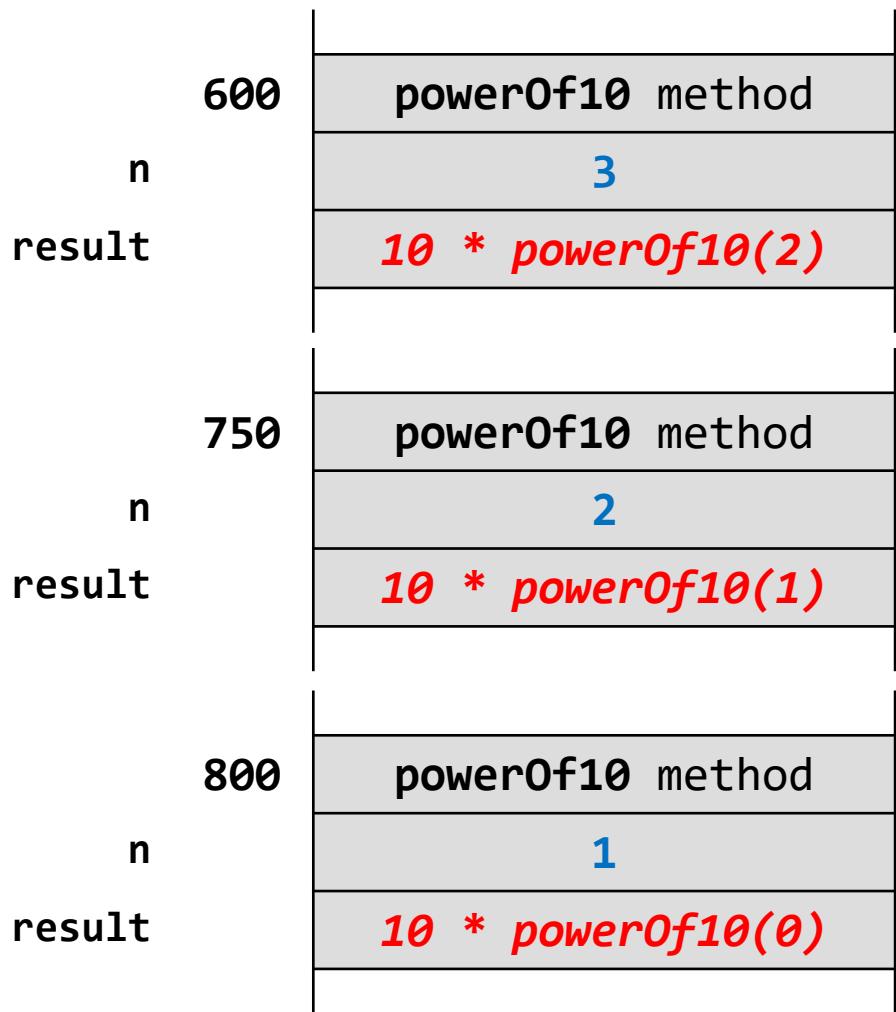
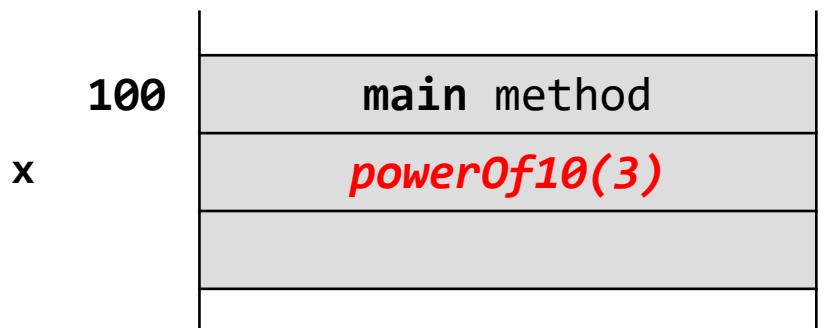
```
double x = Recursion.powerOf10(3);
```



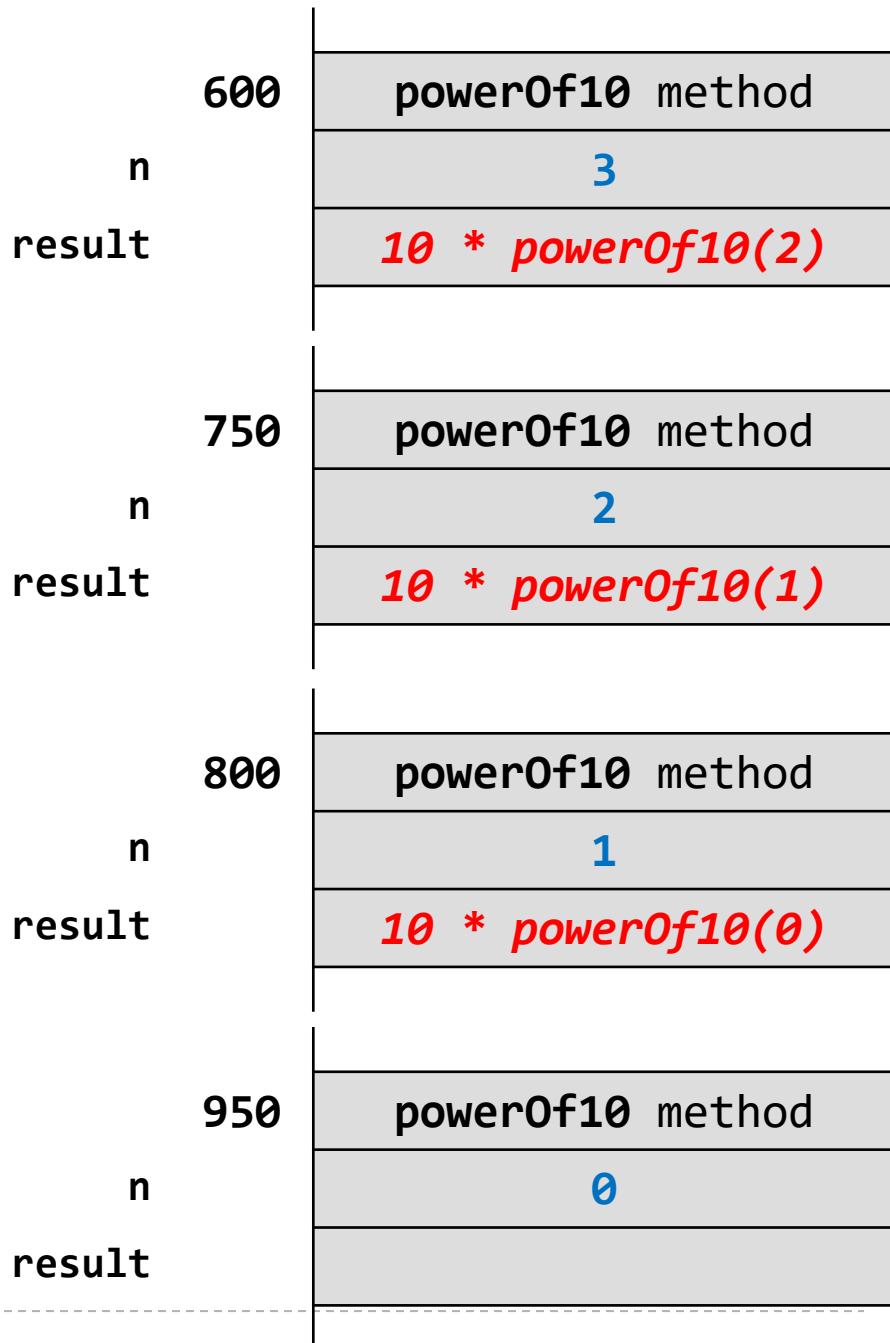
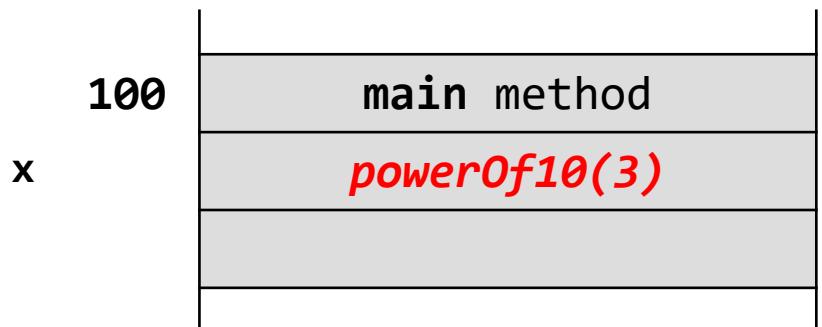
```
double x = Recursion.powerOf10(3);
```



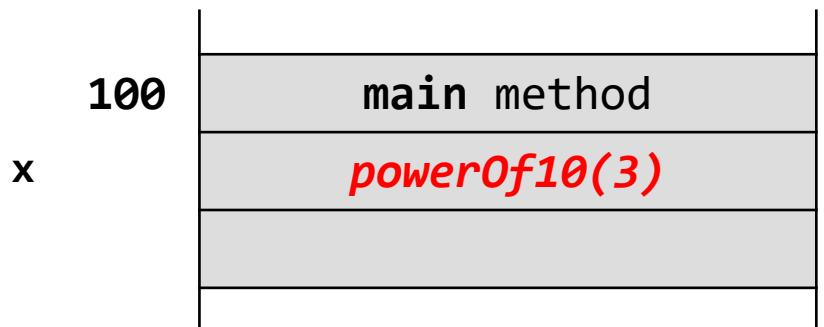
```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```

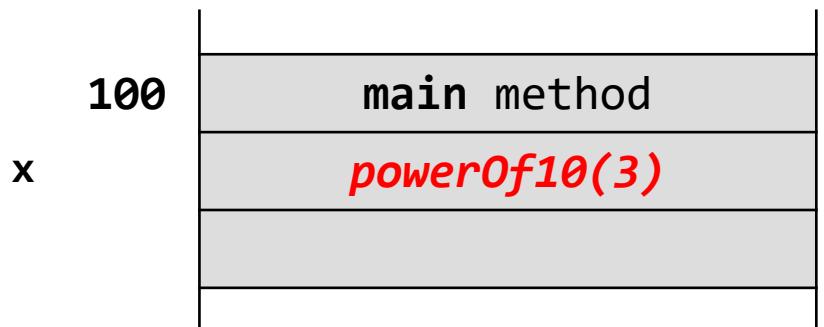


```
double x = Recursion.powerOf10(3);
```



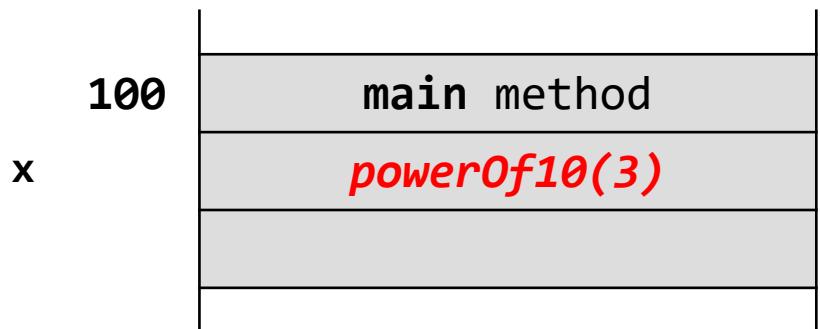
n	600	powerOf10 method
result	3	
	10 * powerOf10(2)	
n	750	powerOf10 method
result	2	
	10 * powerOf10(1)	
n	800	powerOf10 method
result	1	
	10 * powerOf10(0)	
n	950	powerOf10 method
result	0	
	1	

```
double x = Recursion.powerOf10(3);
```



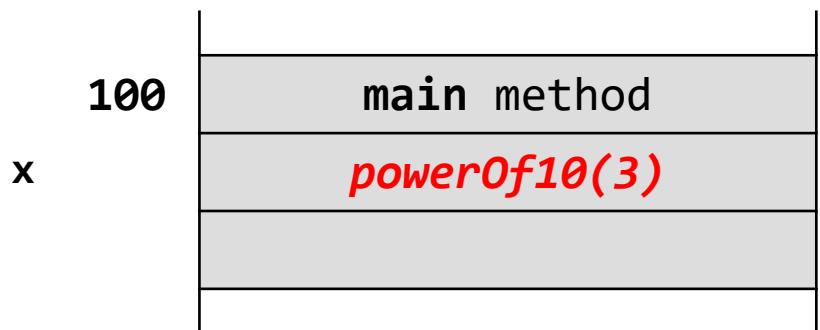
600	n	powerOf10 method
	n	3
	result	$10 * powerOf10(2)$
750	n	powerOf10 method
	n	2
	result	$10 * powerOf10(1)$
800	n	powerOf10 method
	n	1
	result	$10 * 1$
950	n	powerOf10 method
	n	0
	result	1

```
double x = Recursion.powerOf10(3);
```



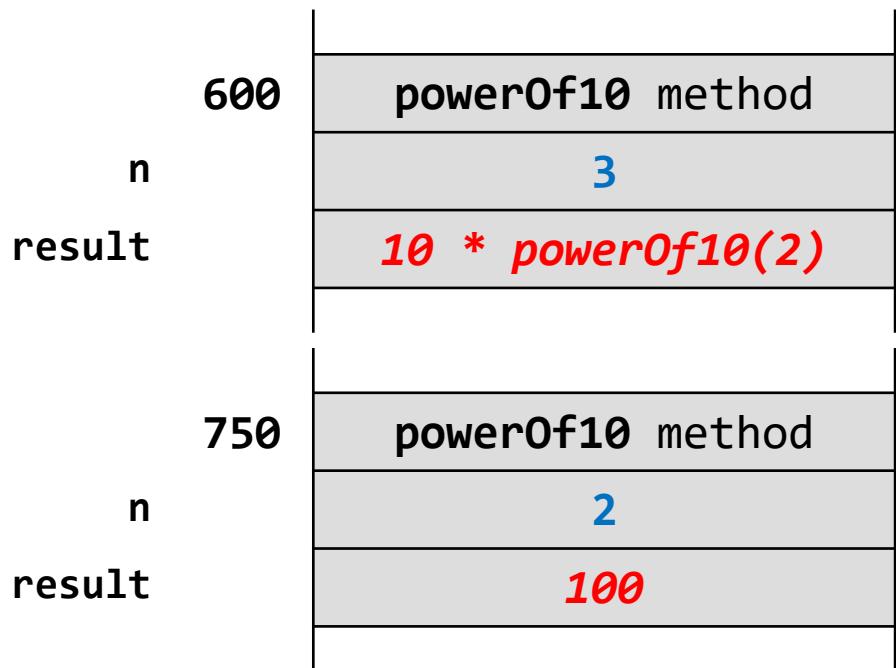
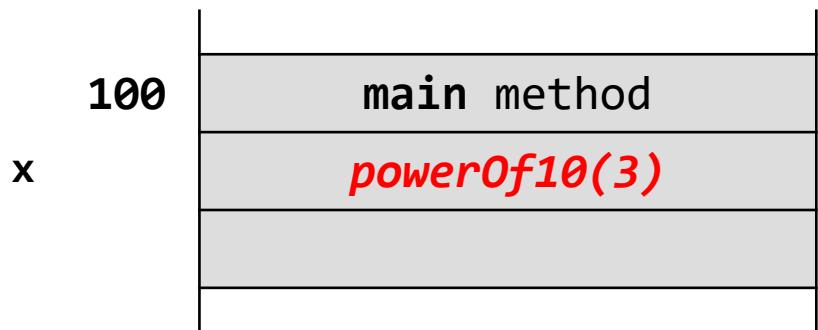
n	600	powerOf10 method
result	3	
		$10 * powerOf10(2)$
n	750	powerOf10 method
result	2	
		$10 * powerOf10(1)$
n	800	powerOf10 method
result	1	
		10

```
double x = Recursion.powerOf10(3);
```

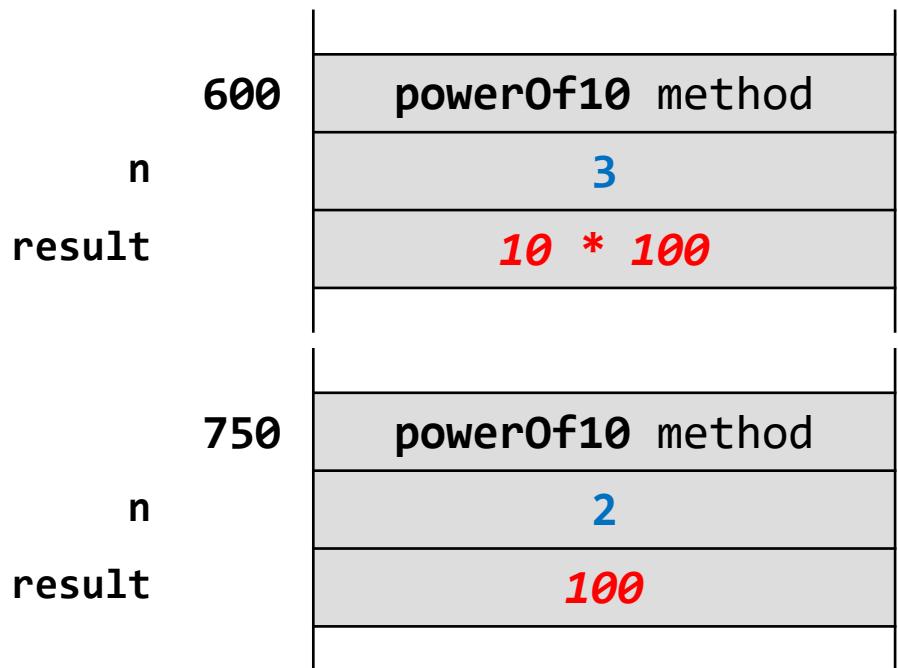
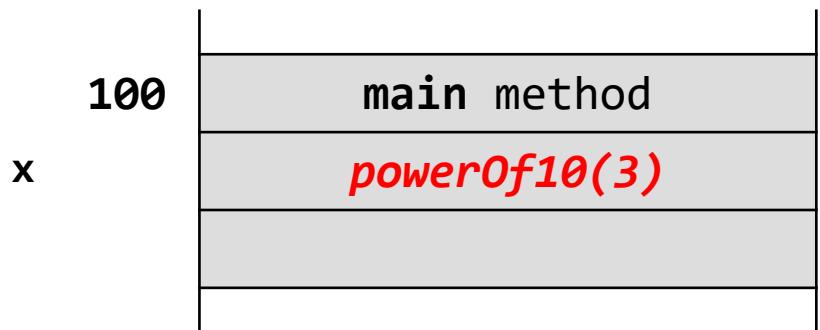


n	600	powerOf10 method
result	3	
	600	$10 * powerOf10(2)$
n	750	powerOf10 method
result	2	
	750	$10 * 10$
n	800	powerOf10 method
result	1	
	800	10

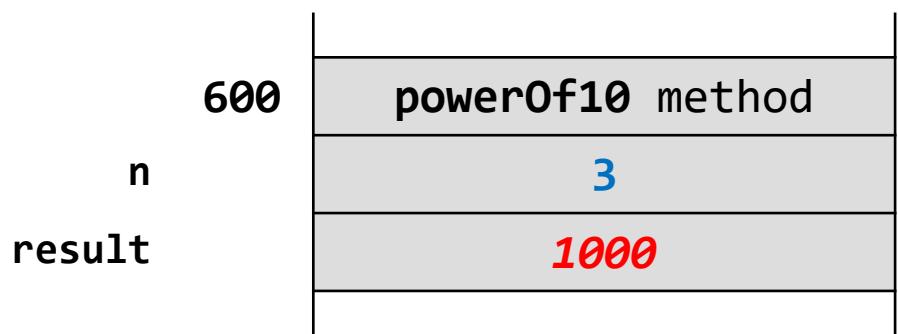
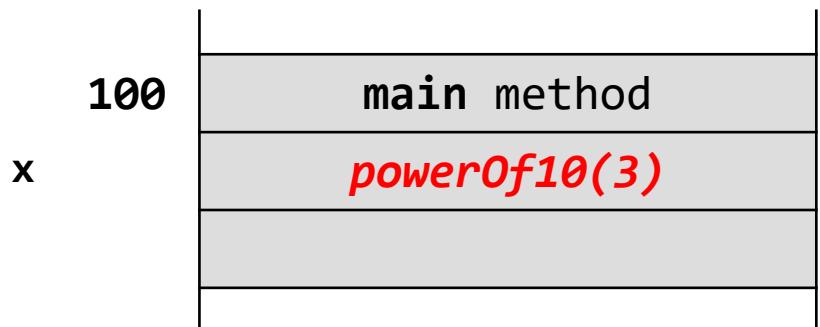
```
double x = Recursion.powerOf10(3);
```



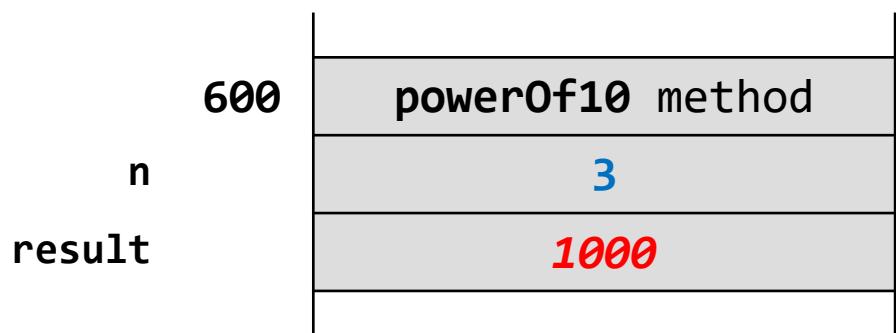
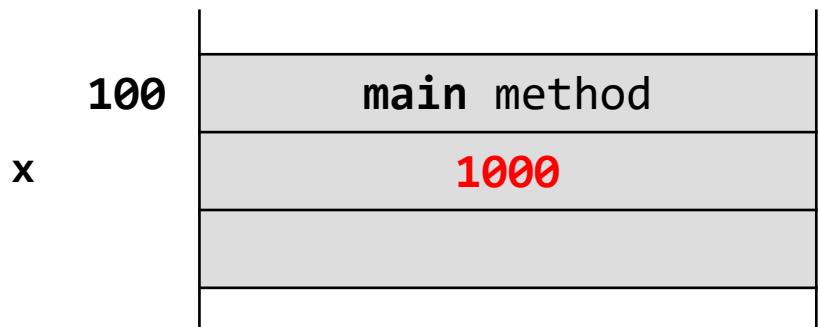
```
double x = Recursion.powerOf10(3);
```



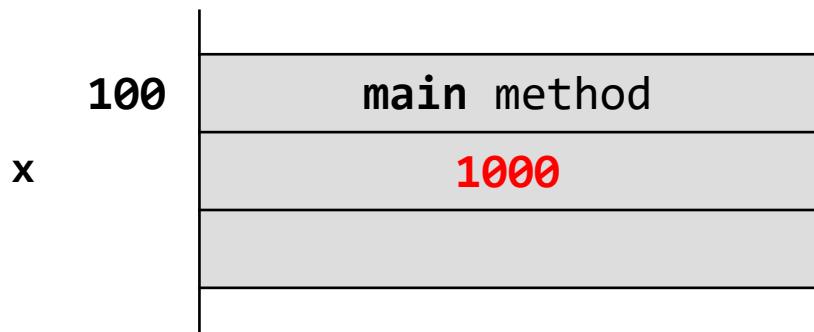
```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```



```
double x = Recursion.powerOf10(3);
```



Using a stack

- ▶ we want to simulate what the JVM is doing during the recursive method
- ▶ the basic idea is to push some information onto a stack to remember what needs to be done when the recursive call returns

-
- ▶ in **powerOf10** what needs to be done when each recursive call returns?
 1. multiply the return value of the recursive call by **10**
 2. multiply the return value of the recursive call by **1**
 3. multiply the return value of the recursive call by **0**
 4. none of the above

Using a stack

- ▶ we want to simulate what the JVM is doing during the recursive method
- ▶ each recursive call that is not a base case pushes a **10** onto the stack

-
- ▶ during each recursive call, what happens to the value n ?
 1. n stays the same
 2. n decreases by 1
 3. n increases by 1
 4. none of the above

```
public static int powerOf10(int n) {  
  
    Stack<Integer> t = new Stack<Integer>();  
    // recursive calls  
    while (n > 0) {  
        t.push(10);  
        n--;  
    }  
  
    ...  
}  
}
```

Using a stack

- ▶ we want to simulate what the JVM is doing during the recursive method
- ▶ each recursive call that is not a base case pushes a **10** onto the stack
- ▶ a recursive call that reaches a base case results in a **1**

```
public static int powerOf10(int n) {  
  
    Stack<Integer> t = new Stack<Integer>();  
    // recursive calls  
    while (n > 0) {  
        t.push(10);  
        n--;  
    }  
    // base case: n == 0  
    int result = 1;  
  
    }  
}
```

Using a stack

- ▶ we want to simulate what the JVM is doing during the recursive method
- ▶ each recursive call that is not a base case pushes a **10** onto the stack
- ▶ a recursive call that reaches a base results in a **1**
- ▶ **pop the stack until it is empty, multiplying the values as you go**

```
public static int powerOf10(int n) {  
  
    Stack<Integer> t = new Stack<Integer>();  
    // recursive calls  
    while (n > 0) {  
        t.push(10);  
        n--;  
    }  
    // base case: n == 0  
    int result = 1;  
  
    // accumulate the result  
    while (!t.isEmpty()) {  
        result = result * t.pop();  
    }  
    return result;  
}
```

Converting a Recursive Method

- ▶ using a stack in the previous example is too complicated for the problem that we were trying to solve
- ▶ but it illustrates the basic idea of using a stack to convert a recursive method to an iterative method

Implementation with Array

- ▶ the **ArrayList** version of stack hints at how to implement a stack using a plain array
 - ▶ however, an array always holds a fixed number of elements
 - ▶ you cannot add to the end of the array without creating a new array
 - ▶ you cannot reduce the size of the array without creating a new array
- ▶ instead of adding and removing from the end of the array, we need to keep track of which element of the array represents the current top of the stack
 - ▶ we need a field for this index

```
import java.util.Arrays;  
  
public class IntStack {  
    // the initial capacity of the stack  
    private static final int DEFAULT_CAPACITY = 16;  
  
    // the array that stores the stack  
    private int[] arr;  
  
    // the index of the top of the stack (equal to -1 for an empty stack)  
    private int topIndex;
```

```
/**  
 * Create an empty stack.  
 */  
public IntStack() {  
    this.arr = new int[IntStack.DEFAULT_CAPACITY];  
    this.topIndex = -1;  
}
```

Implementation with Array

```
IntStack t = new IntStack();
```

this.topIndex == -1

this.arr

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

index

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Implementation with Array

- ▶ pushing a value onto the stack:
 - ▶ increment **this.topIndex**
 - ▶ set the value at **this.arr[this.topIndex]**

Implementation with Array

```
IntStack t = new IntStack();
t.push(7);
```

this.topIndex == 0

this.arr	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Implementation with Array

```
IntStack t = new IntStack();
t.push(7);
t.push(-5);
```

this.topIndex == 1

this.stack	7	-5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Implementation with Array

- ▶ popping a value from the stack:
 - ▶ get the value at **this.arr[this.topIndex]**
 - ▶ decrement **this.topIndex**
 - ▶ return the value

- ▶ notice that we do not need to modify the value stored in the array

Implementation with Array

```
IntStack t = new IntStack();
t.push(7);
t.push(-5);
int value = t.pop(); // value == -5
```

this.topIndex == 0

this.arr	7	-5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

```
/**  
 * Pop and return the top element of the stack.  
 *  
 * @return the top element of the stack  
 * @throws EmptyStackException if the stack is empty  
 */  
public int pop() {  
    // is the stack empty?  
    if (this.topIndex == -1) {  
        throw new EmptyStackException();  
    }  
    // get the element at the top of the stack  
    int element = this.arr[this.topIndex];  
  
    // adjust the top of stack index  
    this.topIndex--;  
  
    // return the element that was on the top of the stack  
    return element;  
}
```

Implementation with Array

```
// stack state when we can safely do one more push
```

this.topIndex == 14

this.arr

7	-5	6	3	2	1	0	0	9	-3	2	7	1	-2	1	0
---	----	---	---	---	---	---	---	---	----	---	---	---	----	---	---

index

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

```
/**  
 * Push an element onto the top of the stack.  
 *  
 * @param element the element to push onto the stack  
 */  
public void push(int element) {  
    // is there capacity for one more element?  
    if (this.topIndex < this.arr.length - 1) {  
        // increment the top of stack index and insert the element  
        this.topIndex++;  
        this.arr[this.topIndex] = element;  
    }  
    else {
```

Adding Capacity

- ▶ if we run out of capacity in the current array we need to add capacity by doing the following:
 - ▶ make a new array with greater capacity
 - ▶ how much more capacity?
 - ▶ copy the old array into the new array
 - ▶ set **this.arr** to refer to the new array
 - ▶ push the element onto the stack

```
else {  
    // make a new array with double previous capacity  
    int[] newArr = new int[this.arr.length * 2];  
  
    // copy the old array into the new array  
    for (int i = 0; i < this.arr.length; i++) {  
        newArr[i] = this.arr[i];  
    }  
  
    // refer to the new array and push the element onto the stack  
    this.arr = newArr;  
    this.push(element);  
}  
}
```

Adding Capacity

- ▶ when working with arrays, it is a common operation to have to create a new larger array when you run out of capacity in the existing array
- ▶ you should use **Arrays.copyOf** to create and copy an existing array into a new array

```
else {  
    int[] newArr = Arrays.copyOf(this.arr, this.arr.length * 2);  
  
    // refer to the new array and push the element onto the stack  
    this.arr = newArr;  
    this.push(element);  
}  
}
```

Stack interface

```
package ca.queensu.cs.cisc124.notes.interfaces;

/**
 * The {@code Stack} interface represents a last-in-first-out (LIFO) stack of
 * strings. In addition to the usual push and pop methods, this interface
 * allows the user to get the number of strings in a stack and to query
 * if the stack is empty.
 */
public interface Stack {

    /**
     * Returns the number of elements in this stack.
     *
     * @return the number of elements in this stack
     */
    public int size();

    /**
     * Returns {@code true} if this stack contains no elements. The default
     * implementation simply returns {@code size() == 0}.
     *
     * @return true if this stack contains no elements
     */
    default boolean isEmpty() {
        return this.size() == 0;
    }
}
```

```
/**  
 * Pushes the specified element on to the top of this stack.  
 *  
 * @param elem the element to push onto this stack  
 */  
public void push(String elem);  
  
/**  
 * Removes the element on the top of this stack and  
 * returns the element.  
 *  
 * @return the top element of this stack  
 * @throws RuntimeException if the stack is empty  
 */  
public String pop();  
}
```

Default interface methods

- ▶ an interface can have *default methods*
 - ▶ classes cannot have default methods
- ▶ a default method has an implementation
 - ▶ the implementation can use only methods from:
 - ▶ the interface itself
 - ▶ **java.lang.Object**
 - ▶ any super-interfaces of the interface
 - e.g., a default method in **List** could use a method from the super-interface **Collection**

Default interface methods

- ▶ default interface methods were added in Java 8 to address the following problem:
 - ▶ it is impractical to add new methods to a widely used interface because every implementing class must be modified to add the new methods
- ▶ Java 8 added a major feature called lambda expressions which required modifying many existing interfaces
 - ▶ the designers of the Java language decided that it was preferable to add default methods to the language rather than break all existing code that implemented the interfaces that needed to be updated

ArrayList-based stack

- ▶ we can easily modify our existing **ArrayList**-based stack so that implements our interface
- ▶ but we can't do the same thing for our array-based stack of **int** because the interface is for a stack of strings
 - ▶ we will fix this when we learn about implementing generic classes and interfaces

```
import java.util.ArrayList;

public class ListStack implements Stack {

    private ArrayList<String> stack;

    public Stack() {
        this.stack = new ArrayList<>();
    }

    public int size() {
        return this.stack.size();
    }

    public void push(String elem) {
        this.stack.add(elem);
    }

    public String pop() {
        String elem = this.stack.remove(this.size() - 1);
        return elem;
    }
}
```

Change class name to ListStack and declare that it implements our Stack interface.

Stacks with linked nodes

Complexity of push and pop

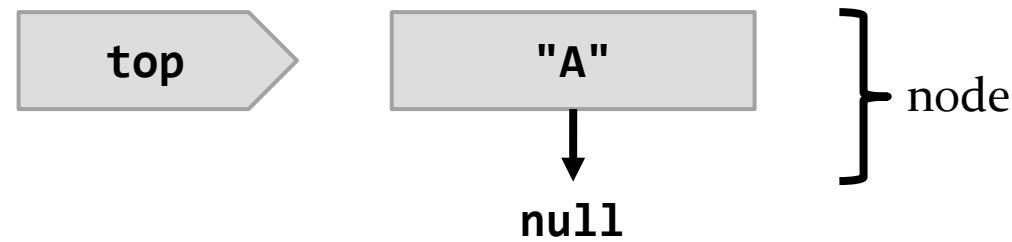
- ▶ for an array-based stack or **ArrayList**-based stack, the complexity of the:
 - ▶ **pop** operation is always in $O(1)$
 - ▶ **push** operation is in $O(N)$ when the stack size equals the array capacity
- ▶ we can guarantee a **push** operation having $O(1)$ complexity by changing how we store elements

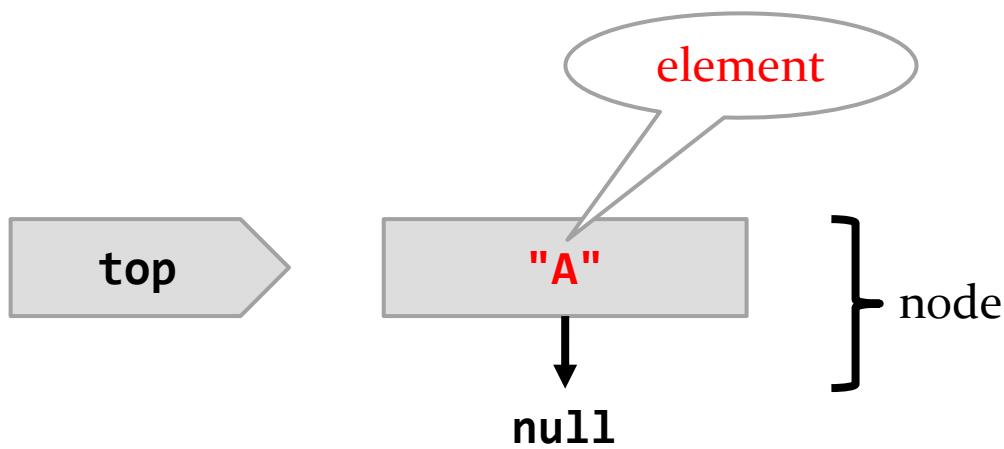
Nodes

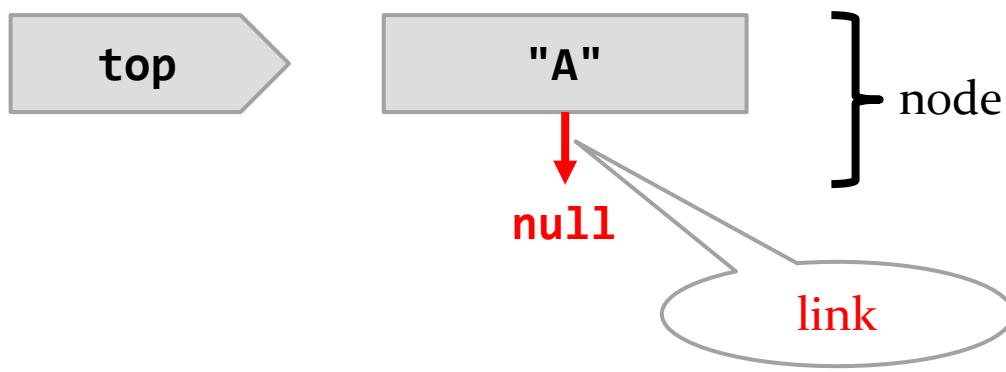
- ▶ many data structures that represent a collection of elements can be implemented using *nodes*
 - ▶ linked lists (CISC124)
 - ▶ trees (CISC235)
 - ▶ graphs (CISC235 (sometimes), CISC365, MATH401)
- ▶ in general, a node stores:
 - ▶ an element (primitive type elements), or a reference to an element
 - ▶ references to zero or more other nodes
 - ▶ these references are often called *links*

Nodes for a stack

- ▶ a stack is a linear collection of elements
 - ▶ elements are arranged in a sequence starting from the top element
 - ▶ each element is connected to the next element deeper in the stack
- ▶ a node in a stack stores:
 - ▶ an element (primitive type elements), or a reference to an element
 - ▶ a reference to the next node deeper in the stack

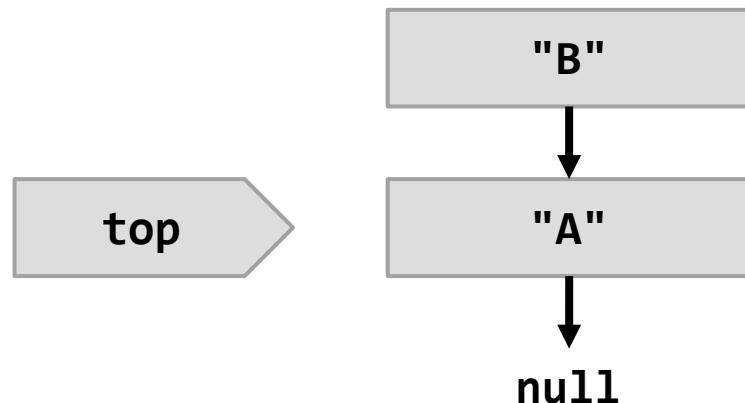


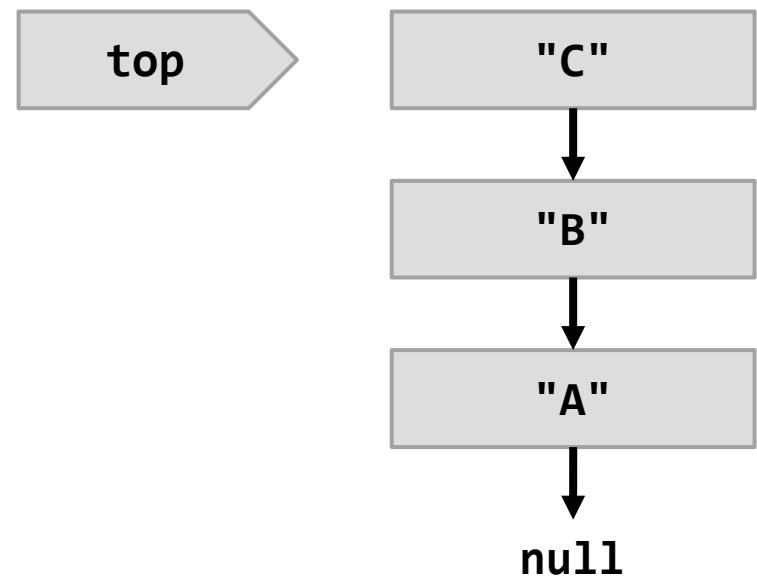




pushing an element onto the stack:

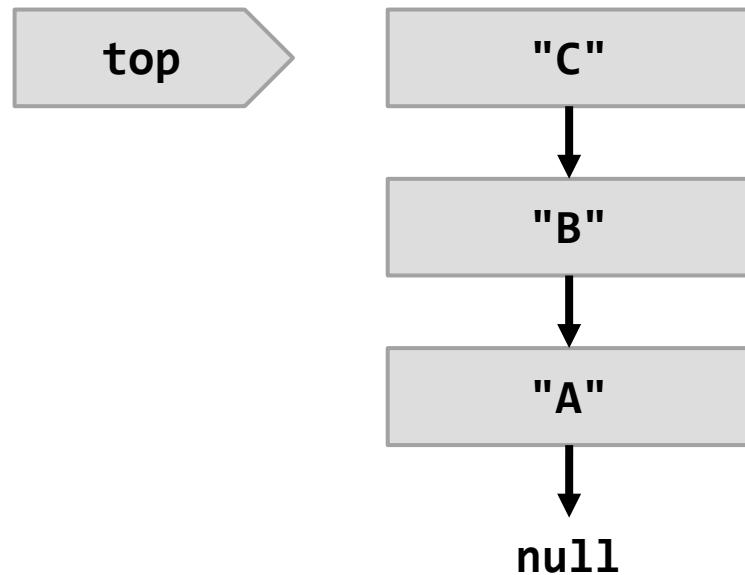
1. make a new node to hold the element pushed onto the stack
2. set the link of the new node to point to the current top node
3. set top to refer to the new node
4. add one to the stack size





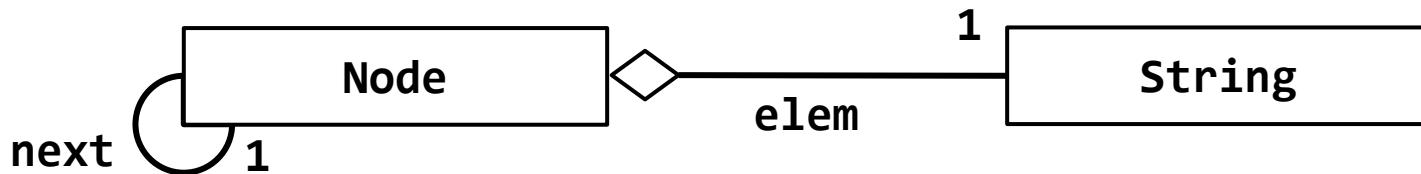
popping an element off the stack:

1. obtain a reference to the element stored in the current top node
2. set top to refer to the next node deeper in the stack
3. subtract one from the stack size
4. return the reference obtained in Step 1 to return the popped element



Node UML diagram

- ▶ a node in a stack stores:
 - ▶ an element (primitive type elements), or a reference to an element
 - ▶ a reference to the next node deeper in the stack
- ▶ for a stack of strings:



```
package ca.queensu.cs.cisc124.notes.interfaces;

/**
 * A linked list-based implementation of the {@code Stack}
 * interface.
 *
 */
public class LinkedStack implements Stack {
    // the number of elements currently on the stack
    private int size;

    // the node containing the top element of the stack
    private Node top;
```

```
private static class Node {  
  
    // the element stored in the node  
    String elem;  
  
    // the link to the next node in the sequence  
    Node next;  
  
    Node(String elem, Node next) {  
        this.elem = elem;  
        this.next = next;  
    }  
}
```

Static member classes

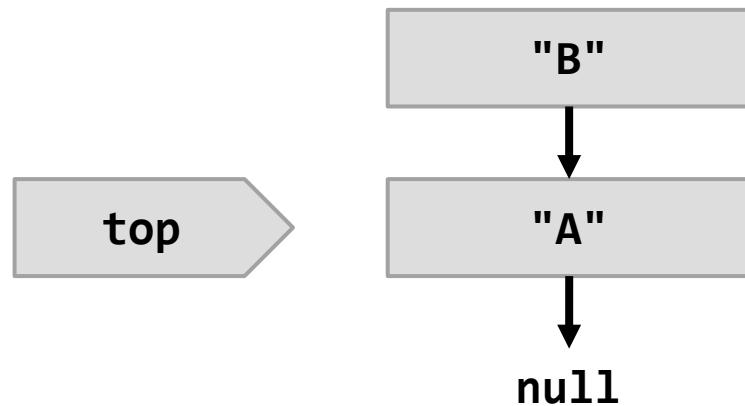
- ▶ the simplest kind of nested class in Java
- ▶ just an ordinary class that happens to be defined inside of another class
- ▶ unlike other kinds of nested classes:
 - ▶ a static member class cannot refer to the private members of the enclosing class
 - ▶ e.g., a **Node** does not have access to the private fields of **LinkedStack**
 - ▶ instances of a static member class can exist in isolation from the enclosing class
 - ▶ e.g., a **Node** object can exist without belonging to a **LinkedStack** object

```
public LinkedStack() {  
    this.size = 0;  
    this.top = null;  
}
```

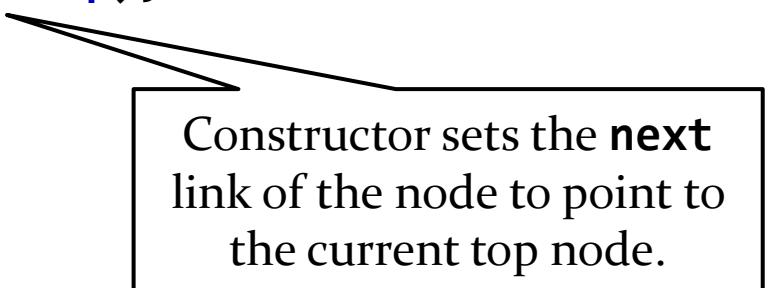
```
@Override  
public int size() {  
    return this.size;  
}
```

pushing an element onto the stack:

1. make a new node to hold the element pushed onto the stack
2. set the link of the new node to point to the current top node
3. set top to refer to the new node
4. add one to the stack size



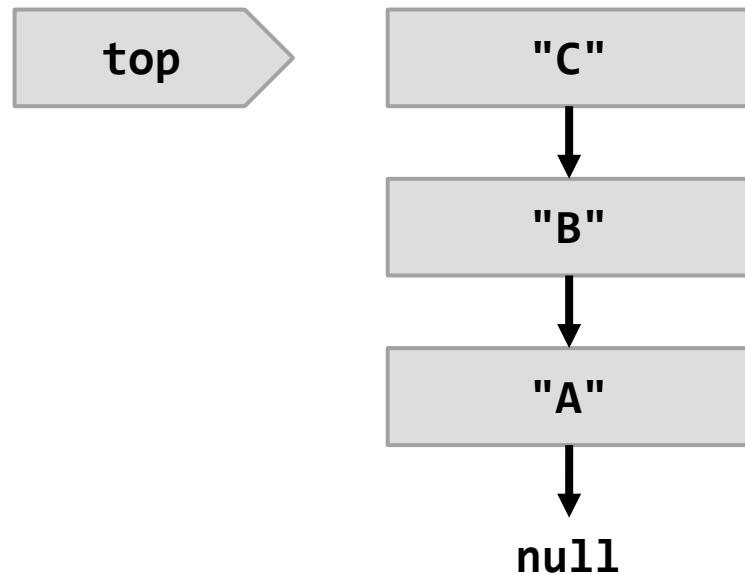
```
@Override  
public void push(String elem) {  
    Node n = new Node(elem, this.top);  
    this.top = n;  
    this.size++;  
}
```



Constructor sets the **next** link of the node to point to the current top node.

popping an element off the stack:

1. obtain a reference to the element stored in the current top node
2. set top to refer to the next node deeper in the stack
3. subtract one from the stack size
4. return the reference obtained in Step 1 to return the popped element



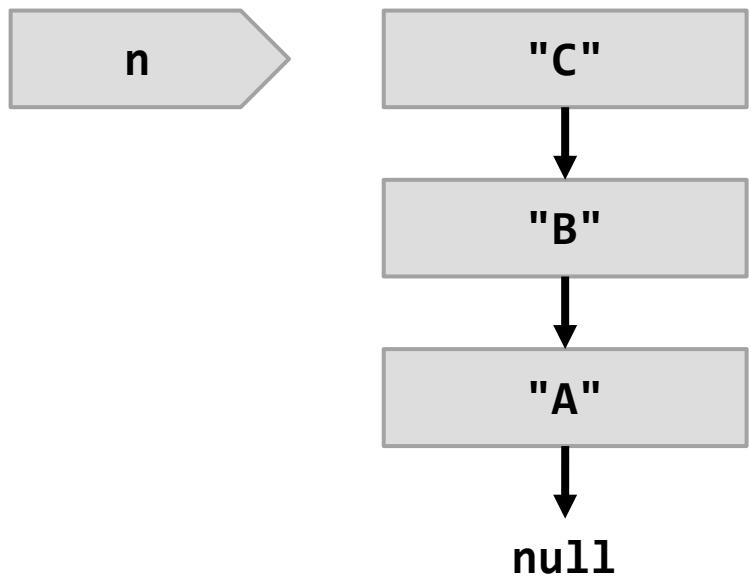
```
@Override
public String pop() {
    if (this.isEmpty()) {
        throw new RuntimeException("popped an empty stack");
    }
    String popped = this.top.elem;
    this.top = this.top.next;
    this.size--;
    return popped;
}
```

Iterating over a linked sequence

- ▶ **toString** for a stack requires iterating over the sequence of nodes
- ▶ iterating over a sequence of linked nodes is a very common operation when working with linked structures
- ▶ observe that the end of the sequence is indicated by a null next link
- ▶ starting at the first node of the sequence, we can simply follow each next link to the next node until we reach a null node

General iteration pattern

```
Node n = startingNode;  
while (n != null) {  
    // do something with n here  
    //  
    // then advance to next node  
    n = n.next;  
}  
// DO NOT use n here!!!
```



```
@Override
public String toString() {
    StringBuilder b = new StringBuilder("Stack:");
    Node n = this.top;
    while (n != null) {
        b.append('\n');
        b.append(n.elem);
        n = n.next;
    }
    return b.toString();
}

} // end LinkedStack
```

Structural recursion

- ▶ the **Node** class is an example of recursion of structure
 - ▶ every **Node** has a reference to a **Node** which has a reference to a **Node** which has a reference to a **Node**, and so on
 - ▶ every **Node** can be viewed as being the top node of a stack
- ▶ the recursive structure makes it easy to write recursive algorithms
 - ▶ e.g., consider **toString()**

```
public String toString() {  
    StringBuilder b = new StringBuilder("Stack:");  
    Node n = this.top;  
    b.append(toStringHelper(n));  
    return b.toString();  
}  
  
private String toStringHelper(Node n) {  
    if (n == null) {  
        return "";  
    }  
    String s = "\n" + n.elem + toStringHelper(n.next);  
    return s;  
}
```

Inheritance

Inheritance in Java

- ▶ inheritance is a relationship between two classes where one class is *derived from* another class

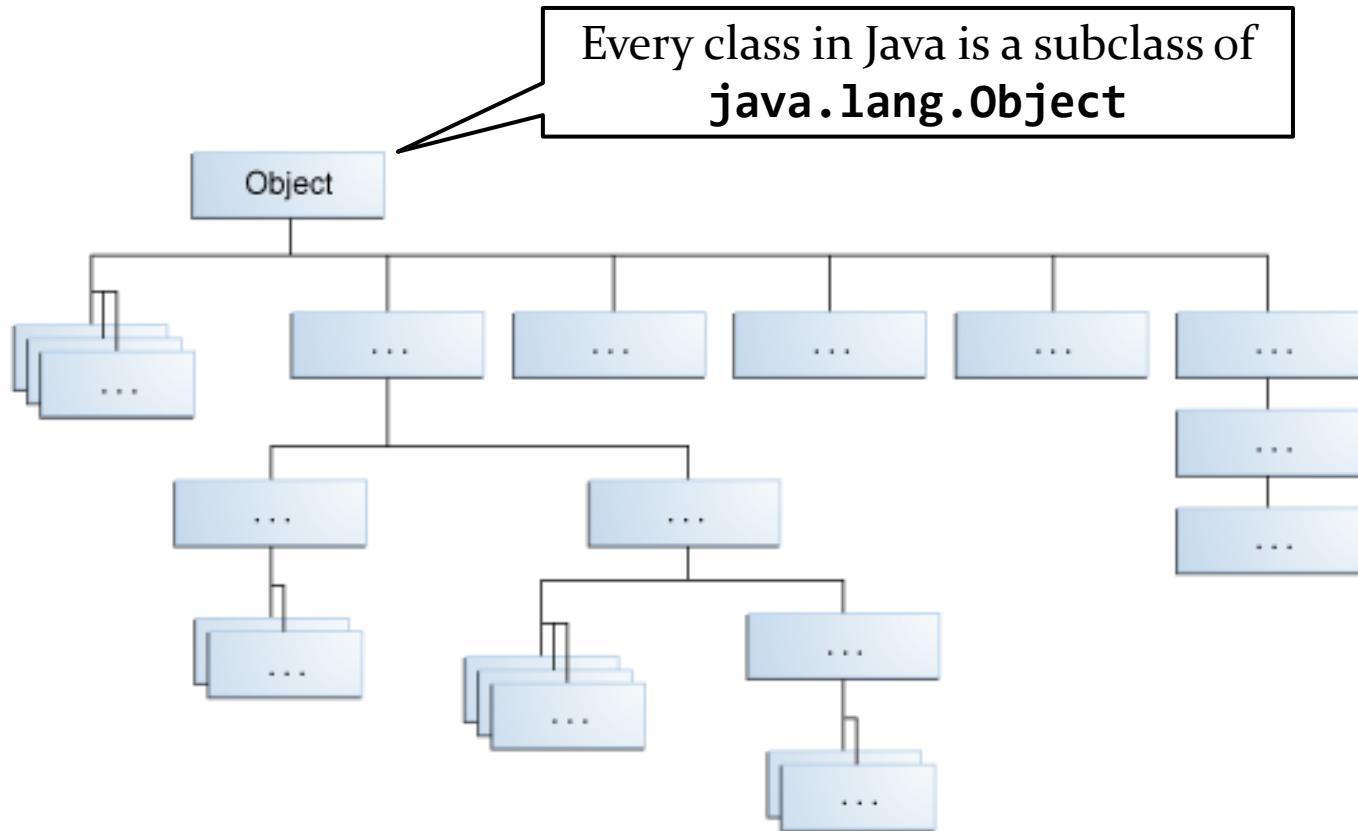
“The idea of inheritance is simple but powerful:
When you want to create a new class and there is already
a class that includes some of the code that you want, you
can derive your new class from the existing class. In doing
this, you can reuse the fields and methods of the existing
class without having to write (and debug!) them yourself.”

<https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html>

Object model

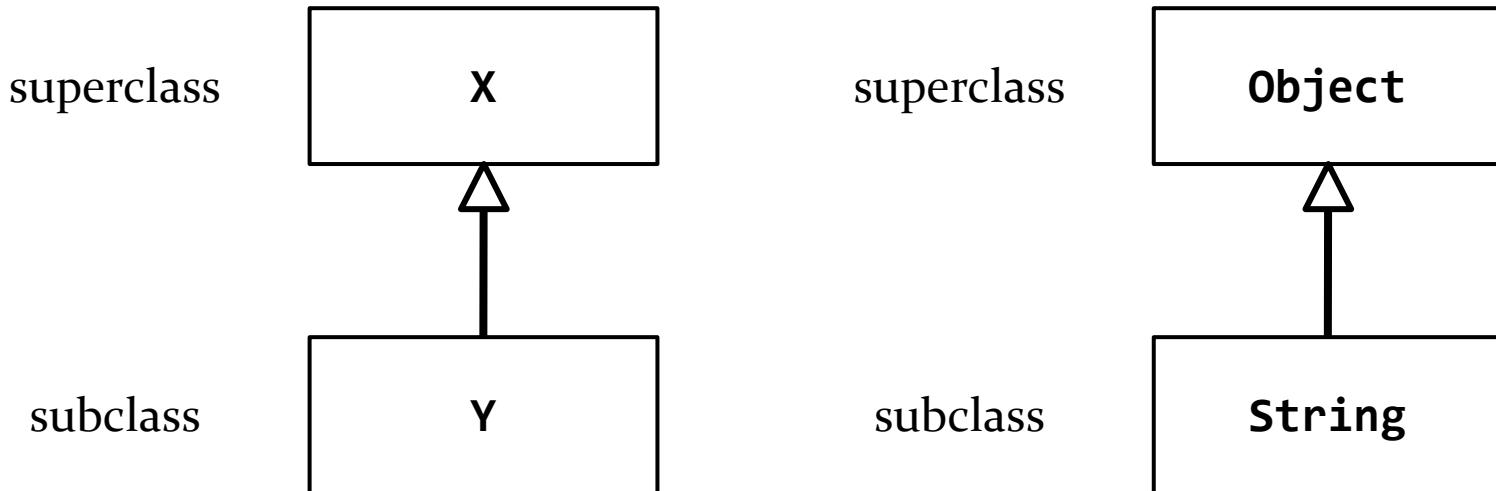
1. abstraction
2. encapsulation
3. modularity
4. hierarchy
 - ▶ interfaces and inheritance are Java mechanisms for supporting a hierarchy of classes

Inheritance in Java



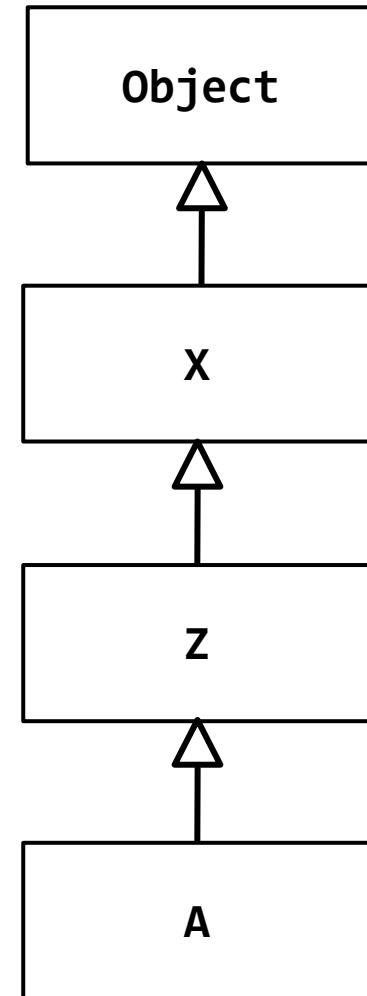
Inheritance in Java

- ▶ subclass (or derived class, extended class, child class)
 - ▶ a class that is derived from another class
- ▶ superclass (or base class, parent class)
 - ▶ the class from which a subclass is derived



Inheritance in Java

- ▶ a class can be derived from a class that is derived from a class, and so on, all the way back to `java.lang.Object`
- ▶ a class is said to be *descended* from all of the classes in the inheritance chain going back to `Object`
- ▶ all of the classes that a class is descended from are called the *ancestors*



Why Inheritance?

- ▶ a subclass inherits all of the non-private members (fields and methods ***but not constructors***) from its superclass
- ▶ if there is an existing class that provides some of the functionality you need you can derive a new class from the existing class
- ▶ the new class has direct access to the **public** and **protected** fields and methods without having to re-declare or re-implement them
- ▶ the new class can introduce new fields and methods
- ▶ the new class can re-define (override) its superclass methods

Is-A

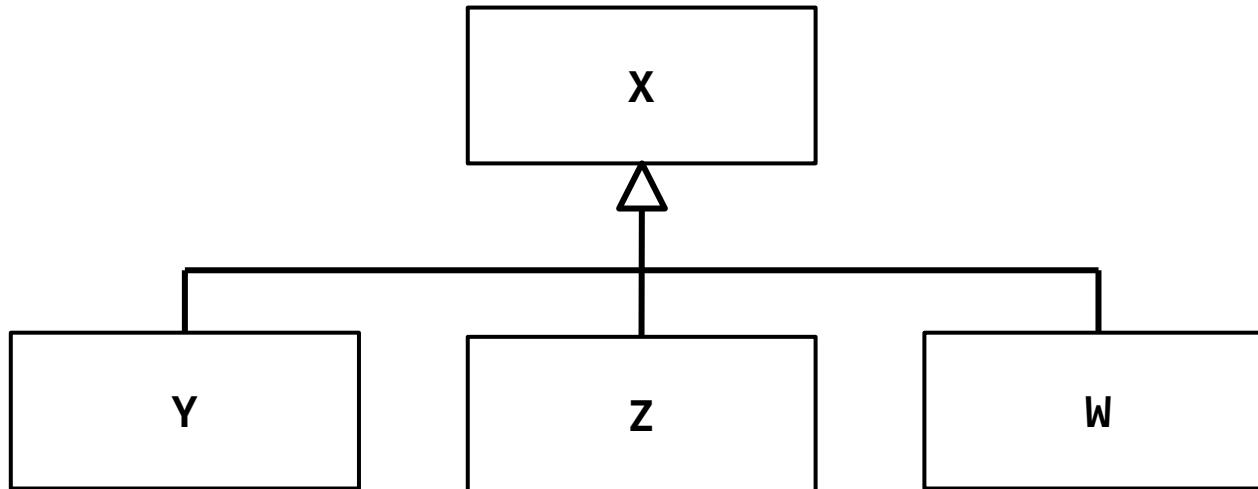
- ▶ inheritance models the *is-a* relationship between classes
- ▶ *is-a* means that a subclass *is-substitutable-for* any of its ancestor classes
 - ▶ e.g., a **String** object is substitutable for an **Object** object
 - a **String** object can do anything that an **Object** object can do

Inheritance in Java

- ▶ in Java, the class **java.lang.Object** is unique
 - ▶ it is the only class that has no superclass
 - ▶ it is the class from which all other classes are descended from
- ▶ if you create a new class and do not explicitly state what the superclass is then the superclass for your new class is **java.lang.Object**

Inheritance in Java

- ▶ in Java a superclass can have many subclasses
- ▶ in Java a subclass can have only one superclass
 - ▶ called *single inheritance*



Inheritance in Java

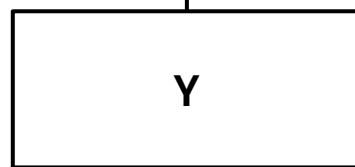
- ▶ single inheritance prevents the deadly diamond of death problem that occurs in programming languages that allow for multiple inheritance

If **X** defines a method **f()**...

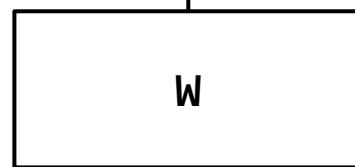


X

... and **Y** and **W** override **f()**...

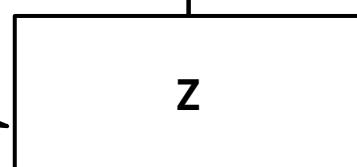


Y



W

... which version of **f()** does **Z** inherit?



Z

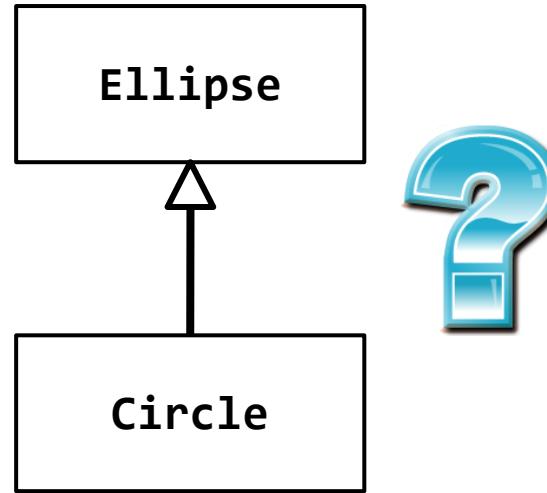
Is-A

- ▶ from a Java point of view, is-a, or is-substitutable-for, means you can use a derived class instance in place of an ancestor class instance
- ▶ for example, suppose that you have a method with one parameter of type **Object**

```
public SomeClass {  
    public static someMethod(Object obj) {  
        // does something with obj  
    }  
}  
  
// client code of someMethod  
  
String s = "hello";  
SomeClass.someMethod(s);           // ok, String is-a Object  
HashSet<Double> t = new HashSet<>();  
SomeClass.someMethod(t);          // ok, HashSet is-a Object  
Point2 p = new Point2();  
SomeClass.someMethod(t);          // ok, Point2 is-a Object
```

Is-A Pitfalls

- ▶ is-a has nothing to do with the real world
- ▶ is-a has everything to do with how the implementer has modelled the inheritance hierarchy
- ▶ the classic example:
 - ▶ **Circle** is-a **Ellipse**?



Circle is-a **Ellipse**?

- ▶ mathematically a circle is a kind of ellipse
- ▶ *but if **Ellipse** can do something that **Circle** cannot, then **Circle** is-a **Ellipse** is false for the purposes of inheritance*
- ▶ remember: is-a means you can substitute a derived class instance for one of its ancestor instances
 - ▶ if **Circle** cannot do something that **Ellipse** can do then you cannot (safely) substitute a **Circle** instance for an **Ellipse** instance

```
// method in Ellipse

/**
 * Changes the width and height of the ellipse to the
 * specified positive width and positive height.
 *
 * @param width the desired width
 * @param height the desired height
 */
public void setSize(double width, double height) {
    this.width = width;
    this.height = height;
}
```

Circle is-a Ellipse?

- ▶ the contract for **setSize** says that the width and height of the shape will be changed to the specified width and height
- ▶ if you do this for a circle then there is no guarantee that the resulting shape is still a circle

Circle is-a **Ellipse**?

- ▶ what if there is no **setSize** method?
 - ▶ if a **Circle** can do everything an **Ellipse** can do then **Circle** can be derived from **Ellipse**

A Naïve Inheritance Example

- ▶ a stack is an important data structure in computer science
- ▶ data structure: an organization of information for better algorithm efficiency or conceptual unity
 - ▶ e.g., list, set, map, array
- ▶ widely used in computer science and computer engineering
 - ▶ e.g., undo/redo can be implemented using two stacks

Implementing stack using inheritance

- ▶ a stack looks a lot like a list
 - ▶ pushing an element onto the top of the stack looks like adding an element to the end of a list
 - ▶ popping an element from the top of a stack looks like removing an element from the end of the list
- ▶ if we have stack inherit from list, our stack class inherits the **add** and **remove** methods from list
 - ▶ we don't have to implement them ourselves
- ▶ let's try making a stack of integers by inheriting from **ArrayList<Integer>**

Implementing stack using inheritance

```
import java.util.ArrayList;  
  
public class BadStack extends ArrayList<Integer> {  
}
```

use the keyword **extends**
followed by the name of
the class that you want
to extend

Implementing stack using inheritance

```
import java.util.ArrayList;

public class BadStack extends ArrayList<Integer> {

    public void push(int value) {
        this.add(value);
    }

    public int pop() {
        int last = this.remove(this.size() - 1);
        return last;
    }
}
```

Uses the inherited `add` method from `ArrayList`

Uses the inherited `remove` method from `ArrayList`

Implementing stack using inheritance

- ▶ that's it, we're done!

```
public static void main(String[] args) {  
    BadStack t = new BadStack();  
    t.push(0);  
    t.push(1);  
    t.push(2);  
    System.out.println(t);  
    System.out.println("pop: " + t.pop());  
    System.out.println("pop: " + t.pop());  
    System.out.println("pop: " + t.pop());  
}  
}
```

```
[0, 1, 2]  
pop: 2  
pop: 1  
pop: 0
```

Implementing stack using inheritance

- ▶ why is this a poor implementation?
- ▶ by having **BadStack** inherit from **ArrayList<Integer>** we are saying that a stack is a list
 - ▶ anything a list can do, a stack can also do, such as:
 - ▶ get an element from the middle of the stack (instead of only from the top of the stack)
 - ▶ set an element in the middle of the stack
 - ▶ remove any element of the stack
 - ▶ iterate over the elements of the stack

Implementing stack using inheritance

```
public static void main(String[] args) {  
    BadStack t = new BadStack();  
    t.push(100);  
    t.push(200);  
    t.push(300);  
    System.out.println("get(1)?: " + t.get(1));  
    t.set(1, -1000);  
    System.out.println("set(1, -1000)?: " + t);  
}
```

```
[100, 200, 300]  
get(1)?: 200  
set(1, -1000)?: [100, -1000, 300]
```

Implementing stack using inheritance

- ▶ using inheritance to implement a stack is an example of an incorrect usage of inheritance
- ▶ inheritance should only be used when an is-a relationship exists
 - ▶ a stack is not a list
 - ▶ therefore, we should not use inheritance to implement a stack
- ▶ even experts sometimes get this wrong
 - ▶ early versions of the Java class library provided a stack class that inherited from a list-like class
 - ▶ **java.util.Stack**

Other ways to implement stack

- ▶ use composition (which we've already done)
 - ▶ **Stack has-a List**

Inheritance for code reuse

- ▶ recall the counter example from early in the course:
 - ▶ a counter starts counting from zero
 - ▶ a user can ask for the counter's value
 - ▶ a user can advance the counter upwards by one
 - ▶ when the counter reaches `Integer.MAX_VALUE` advancing the counter causes the counter to wrap around to zero
- ▶ what if you want some other behavior when the counter reaches `Integer.MAX_VALUE`?

```
public class Counter {  
  
    private int value;  
  
    /**  
     * Initializes this counter so that its current value is 0.  
     */  
    public Counter() {  
        this.value = 0;  
    }  
  
    /**  
     * Returns the current value of this counter.  
     *  
     * @return the current value of this counter  
     */  
    public int value() {  
        return this.value;  
    }  
}
```

```
/**  
 * Increment the value of this counter upwards by 1. If this  
 * method is called when the current value of this counter is  
 * equal to {@code Integer.MAX_VALUE} then the value of this  
 * counter is set to 0 (i.e., the counter wraps around to 0).  
 */  
public void advance() {  
    if (this.value != Integer.MAX_VALUE) {  
        this.value++;  
    }  
    else {  
        this.value = 0;  
    }  
}
```

Inheritance for code reuse

- ▶ what if you want some other behavior when the counter reaches `Integer.MAX_VALUE`?
- ▶ if we extend the `Counter` class, we can override the behavior of the `advance` method
 - ▶ but this does not solve the problem because the field `value` in `Counter` is private and there are no methods that can set the value to a specified value
 - ▶ our subclass has no way to modify the value of the counter
 - ▶ this is a common problem when trying to extend a class that was not designed for inheritance

Inheritance for code reuse

- ▶ to extend the **Counter** class we have to make the field **value** accessible to subclasses
- ▶ the **protected** access modifier allows subclasses to access a field defined in a superclass

```
public class Counter {  
  
    protected int value;  
  
    /**  
     * Initializes this counter so that its current value is 0.  
     */  
    public Counter() {  
        this.value = 0;  
    }  
}
```

```
/**  
 * Initializes this counter to the specified non-negative  
 * value.  
  
 * @param value the starting value of this counter  
 * @throws IllegalArgumentException if value is negative  
 */  
public Counter(int value) {  
    if (value < 0) {  
        throw new IllegalArgumentException();  
    }  
    this.value = value;  
}
```

Inheritance for code reuse

- ▶ we should also modify the contract of the advance method to indicate that subclasses might change the behavior of the method

```
/**  
 * Increment the value of this counter upwards by 1. If this  
 * method is called when the current value of this counter is  
 * equal to {@code Integer.MAX_VALUE} then the value of this  
 * counter is set to 0 (i.e., the counter wraps around to 0)  
 * but subclasses can override this behaviour.  
 */  
  
public void advance() {  
    if (this.value != Integer.MAX_VALUE) {  
        this.value++;  
    }  
    else {  
        this.value = 0;  
    }  
}
```

Inheritance for code reuse

- ▶ why should we modify the contract of **advance**?
 - ▶ because subclasses are supposed to be substitutable for the superclass
 - ▶ if we do not modify the contract then there is no way for subclasses to change the behavior of **advance** and still be substitutable

Inheritance for code reuse

- ▶ now we can extend **Counter** to implement different behavior when the counter value reaches **Integer.MAX_VALUE**
- ▶ for example, we can create a counter that stops counting when its value reaches **Integer.MAX_VALUE**

```
public class StoppingCounter extends Counter {  
  
    // no fields!
```

Inheritance for code reuse

- ▶ the **StoppingCounter** class has no fields of its own
 - ▶ the current value of a **StoppingCounter** is stored in a field that belongs to the superclass
 - ▶ there is no need for **StoppingCounter** to add a new field to store the current value
- ▶ a subclass is allowed to add new fields
 - ▶ these fields are not visible to the superclass

```
public class StoppingCounter extends Counter {  
  
    /**  
     * Initializes this counter so that its current value is 0.  
     */  
    public StoppingCounter() {  
        // what goes here?  
    }  
}
```

Constructors of Subclasses

- ▶ the purpose of a constructor is to initialize the value of the fields of **this** object
- ▶ how can a constructor set the value of a field that belongs to the superclass?
 - ▶ by calling the superclass constructor and passing **this** as an implicit argument
 - ▶ works even if the superclass field is **private**

```
public class StoppingCounter extends Counter {  
  
    /**  
     * Initializes this counter so that its current value is 0.  
     */  
    public StoppingCounter() {  
        super();  
    }  
}
```

Constructors of Subclasses

1. the first line in the body of every constructor ***must*** be a call to another constructor
 - ▶ if it is not then Java will insert a call to the superclass default constructor
 - ▶ if the superclass default constructor does not exist or is private then a compilation error occurs
2. a call to another constructor can only occur on the first line in the body of a constructor
3. a superclass constructor must be called during construction of the derived class
 - ▶ any superclass constructor can be called (not just the no-argument constructor)

Inheritance for code reuse

- ▶ we can add a second constructor that initializes the counter to a specified non-negative value

```
/**  
 * Initializes this counter to the specified non-negative value.  
 *  
 * @param value  
 *         the starting value of this counter  
 * @throws IllegalArgumentException  
 *         if value is negative  
 */  
  
public StoppingCounter(int value) {  
    // what goes here?  
}
```

Inheritance for code reuse

- ▶ note that you can use constructor chaining in the subclass
 - ▶ the first line of a constructor should be another constructor call but it does not have to be a call to a superclass constructor
 - ▶ all that is required is that a superclass constructor is eventually called
- ▶ for example, the two constructors of **StoppingCounter** could be correctly implemented as follows

```
public class StoppingCounter extends Counter {  
  
    /**  
     * Initializes this counter so that its current value is 0.  
     */  
    public StoppingCounter() {  
        this(0); // calls the constructor below  
    }  
  
    /**  
     * Initializes this counter to the specified non-negative value.  
     *  
     * @param value  
     *         the starting value of this counter  
     * @throws IllegalArgumentException  
     *         if value is negative  
     */  
    public StoppingCounter(int value) {  
        super(value); // calls the superclass constructor  
    }  
}
```

What is a Subclass?

- ▶ a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and fields
- ▶ inheritance does more than copy the API of the superclass
 - ▶ the subclass contains a subobject of the parent class
 - ▶ the superclass subobject needs to be constructed (just like a regular object)
 - ▶ the mechanism to perform the construction of the superclass subobject is to call the superclass constructor

-
- ▶ why is the constructor call to the superclass needed?
 - ▶ because **StoppingCounter** is-a **Counter** and the **Counter** part of **StoppingCounter** needs to be constructed

```
StoppingCounter c = new Counter(1);
```

1. **StoppingCounter** constructor starts running
 - initializes new **Counter** subobject by invoking the **Counter** constructor
2. **Counter** constructor starts running
 - initializes new **Object** subobject by (silently) invoking the **Object** constructor
3. **Object** constructor runs
 - and finishes
 - sets **value**
 - and finishes
- finishes

StoppingCounter object

Counter object

Object object

value

1

Inheritance for code reuse

- ▶ we can now complete the **StoppingCounter** implementation by overriding **advance**

```
/**  
 * Increment the value of this counter upwards by 1. If this method is  
 * called when the current value of this counter is equal to  
 * {@code Integer.MAX_VALUE} then the value of this counter remains  
 * {@code Integer.MAX_VALUE} (i.e., the counter stops counting  
 * at {@code Integer.MAX_VALUE}).  
 */  
  
@Override  
public void advance() {  
    if (this.value != Integer.MAX_VALUE) {  
        this.value++;  
    }  
}
```

Inheritance for code reuse

- ▶ suppose that we want a counter that throws an exception when its value reaches `Integer.MAX_VALUE`
- ▶ it turns out that this is not a good idea
 - ▶ revisit this when we discuss how postconditions interact with inheritance

```
public class ThrowingCounter extends Counter {  
    // no fields!
```

```
/**  
 * Initializes this counter so that its current value is 0.  
 */  
public ThrowingCounter() {  
    super(0);  
}  
  
/**  
 * Initializes this counter to the specified non-negative value.  
 *  
 * @param value  
 *         the starting value of this counter  
 * @throws IllegalArgumentException  
 *         if value is negative  
 */  
public ThrowingCounter(int value) {  
    super(value);  
}
```

```
/**  
 * Increment the value of this counter upwards by 1. If this method is  
 * called when the current value of this counter is equal to  
 * {@code Integer.MAX_VALUE} then a {@code RuntimeException} is thrown.  
 *  
 * @throws RuntimeException  
 *         if this method is called when the counter is at its  
 *         maximum value  
 */  
  
@Override  
public void advance() {  
    if (this.value != Integer.MAX_VALUE) {  
        this.value++;  
    } else {  
        throw new RuntimeException();  
    }  
}  
}
```

Alternative to inheritance

- ▶ notice that the counters differ only in what happens when the counter is advanced past its maximum value
 - ▶ using inheritance, we created separate classes that override the **advance** method
- ▶ an alternative approach is to encapsulate the behavior of what happens when the counter is advanced past its maximum value in an object

What is a Subclass?

- ▶ a subclass looks like a new class that has the same API as its superclass with perhaps some additional methods and fields
- ▶ inheritance does more than copy the API of the superclass
 - ▶ the subclass contains a subobject of the parent class
 - ▶ the superclass subobject needs to be constructed (just like a regular object)
 - ▶ the mechanism to perform the construction of the superclass subobject is to call the superclass constructor

-
- ▶ why is the constructor call to the superclass needed?
 - ▶ because **StoppingCounter** is-a **Counter** and the **Counter** part of **StoppingCounter** needs to be constructed

```
StoppingCounter c = new Counter(1);
```

1. **StoppingCounter** constructor starts running
 - initializes new **Counter** subobject by invoking the **Counter** constructor
 2. **Counter** constructor starts running
 - initializes new **Object** subobject by (silently) invoking the **Object** constructor
 3. **Object** constructor runs
 - and finishes
 - sets **value**
 - and finishes
- finishes

StoppingCounter object

Counter object

Object object

value

1

Inheritance for code reuse

- ▶ we can now complete the **StoppingCounter** implementation by overriding **advance**

```
/**  
 * Increment the value of this counter upwards by 1. If this method is  
 * called when the current value of this counter is equal to  
 * {@code Integer.MAX_VALUE} then the value of this counter remains  
 * {@code Integer.MAX_VALUE} (i.e., the counter stops counting  
 * at {@code Integer.MAX_VALUE}).  
 */  
  
@Override  
public void advance() {  
    if (this.value != Integer.MAX_VALUE) {  
        this.value++;  
    }  
}
```

Inheritance for code reuse

- ▶ suppose that we want a counter that throws an exception when its value reaches `Integer.MAX_VALUE`
- ▶ it turns out that this is not a good idea
 - ▶ revisit this when we discuss how postconditions interact with inheritance

```
public class ThrowingCounter extends Counter {  
    // no fields!
```

```
/**  
 * Initializes this counter so that its current value is 0.  
 */  
public ThrowingCounter() {  
    super(0);  
}  
  
/**  
 * Initializes this counter to the specified non-negative value.  
 *  
 * @param value  
 *         the starting value of this counter  
 * @throws IllegalArgumentException  
 *         if value is negative  
 */  
public ThrowingCounter(int value) {  
    super(value);  
}
```

```
/**  
 * Increment the value of this counter upwards by 1. If this method is  
 * called when the current value of this counter is equal to  
 * {@code Integer.MAX_VALUE} then a {@code RuntimeException} is thrown.  
 *  
 * @throws RuntimeException  
 *         if this method is called when the counter is at its  
 *         maximum value  
 */  
  
@Override  
public void advance() {  
    if (this.value != Integer.MAX_VALUE) {  
        this.value++;  
    } else {  
        throw new RuntimeException();  
    }  
}  
}
```

Alternative to inheritance

- ▶ notice that the counters differ only in what happens when the counter is advanced past its maximum value
 - ▶ using inheritance, we created separate classes that override the **advance** method
- ▶ an alternative approach is to encapsulate the behavior of what happens when the counter is advanced past its maximum value in an object

equals

- ▶ can we compare the different kinds of counters for equality?
- ▶ it depends on how we implemented **equals** in the super class **Counter**
- ▶ suppose that we used **getClass()** to implement **equals**
 - ▶ recall that when we do this, we are stating that only objects with the exact same type can be equal

```
// in Counter  
  
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (obj == null) {  
        return false;  
    }  
    if (this.getClass() != obj.getClass()) {  
        return false;  
    }  
    Counter other = (Counter) obj;  
    return this.value == other.value;  
}
```

Counters of same type can be equal

- ▶ both of the following comparisons print true:

```
Counter c1 = new Counter();
Counter c2 = new Counter();
System.out.println(c1.equals(c2));
```

```
StoppingCounter c3 = new StoppingCounter();
StoppingCounter c4 = new StoppingCounter();
System.out.println(c3.equals(c4));
```

Counters of same type can be equal

- ▶ the following comparisons print false because the compared counters are from different classes:

```
Counter c1 = new Counter();
Counter c2 = new StoppingCounter();
System.out.println(c1.equals(c2));
```

```
Counter c3 = new StoppingCounter();
Counter c4 = new ThrowingCounter();
System.out.println(c3.equals(c4));
```

equals and mixed type comparisons

- ▶ in some inheritance hierarchies, comparing objects of different types for equality is sensible
- ▶ in such cases, you must use the **instanceof** version of equals

```
// in Counter  
@Override  
public boolean equals(Object obj) {  
    if (this == obj) {  
        return true;  
    }  
    if (!(obj instanceof Counter)) {  
        return false;  
    }  
    Counter other = (Counter) obj;  
    return this.value == other.value;  
}
```

Returns true if **obj** is from a class that has **Counter** as an ancestor.

Counters of same type can be equal

- ▶ both of the following comparisons print true:

```
Counter c1 = new Counter();
Counter c2 = new Counter();
System.out.println(c1.equals(c2));
```

```
StoppingCounter c3 = new StoppingCounter();
StoppingCounter c4 = new StoppingCounter();
System.out.println(c3.equals(c4));
```

Counters of different types can be equal

- ▶ the following comparisons print true even though the compared counters are from different classes because the counters have the same value:

```
Counter c1 = new Counter();
Counter c2 = new StoppingCounter();
System.out.println(c1.equals(c2));
```

```
Counter c3 = new StoppingCounter();
Counter c4 = new ThrowingCounter();
System.out.println(c3.equals(c4));
```

How many types does an object have?

- ▶ with the exception of instances of **java.lang.Object**, objects have more than one type
- ▶ an object's types include:
 - ▶ its own class type,
 - ▶ all its ancestor class types, and
 - ▶ all of its interface types

How many types does an object have?

```
public class Point2 extends Object  
implements Comparable<Point2>
```

- ▶ has the types:
 - ▶ Point2
 - ▶ Object
 - ▶ Comparable<Point2>

How many types does an object have?

```
public class StoppingCounter extends Counter  
implements Comparable<Counter>
```

- ▶ has the types:
 - ▶ **StoppingCounter**
 - ▶ **Counter**
 - ▶ **Object**
 - ▶ **Comparable<Counter>**

How many types does an object have?

```
public final class String extends Object  
implements  
Serializable, Comparable<String>, CharSequence
```

- ▶ has the types:
 - ▶ **String**
 - ▶ **Object**
 - ▶ **Serializable**
 - ▶ **Comparable<String>**
 - ▶ **CharSequence**

Polymorphism

- ▶ Oxford dictionary definition of polymorphism is:
 - ▶ "the condition of occurring in several different forms"
- ▶ Java has *subtype polymorphism*
 - ▶ subtypes are substitutable for their ancestor types
 - ▶ made possible because the type of an object includes the types of all of its ancestor classes

Declared types

- ▶ when creating a variable, the programmer declares its type:
- ▶ for example:

```
Counter c;
```

declares a variable **c** whose *declared type* is **Counter**

- ▶ using **c** we can store:
 - ▶ the value **null** (no object)
 - ▶ a reference to a **Counter** object
 - ▶ a reference to an object having a type that is a descendent of **Counter**

Declared types

- ▶ the declared type of a variable, field, or parameter determines what operations can be performed
- ▶ for example:

```
Object obj = new ArrayList<String>();  
String s = obj.toString();
```

is legal, but

```
obj.add("hello");
```

is a compile-time error

Declared types

- ▶ the declared type of a variable, field, or parameter determines what operations can be performed
- ▶ for example:

```
Set<String> t = new TreeSet<>();  
t.add("hello");
```

is legal, but

```
String s = t.first();
```

is a compile-time error

Declared types

- ▶ the declared type of a variable, field, or parameter determines what operations can be performed
- ▶ for example:

```
SortedSet<String> t = new TreeSet<>();  
t.add("hello");  
String s = t.first();
```

is legal

Runtime types

- ▶ in an assignment or initialization expression, the type of the value on the right-hand side of `=` is the *runtime type* or *actual type*
- ▶ for example:

```
Counter c;  
c = new StoppingCounter();
```

the declared type of `c` is **Counter** and its runtime type is **StoppingCounter**

Runtime types

- ▶ the runtime type determines which version of a non-static method is called when using a variable, field, or parameter
- ▶ for example:

```
Object obj = new Counter();
String s = obj.toString();
```

causes the **Counter** version of **toString** to be called at runtime (not the **Object** version of **toString**)

Runtime types

- ▶ consider the following **print** method:

```
public class Printer {  
  
    public static void print(Object obj) {  
        System.out.println(obj.toString());  
    }  
  
}
```

- ▶ it is the runtime type of **obj** that determines which version of **toString** is called

```
String s = "hi there";
Printer.print(s);      // String version of toString

ArrayList<Integer> t = new ArrayList<>();
t.add(9);
t.add(1);
t.add(1);
Printer.print(t);      // ArrayList version of toString

Object o = new Date();
Printer.print(o);      // Date version of toString
```

Dynamic dispatch

- ▶ observe that the compiler cannot determine which version of **toString** gets called in the method **print** but still manages to compile the **Printer** class without any issues
- ▶ the actual version of **toString** that is called cannot be determined until runtime
- ▶ using the runtime type to determine which version of a method to call is called *dynamic dispatch* or *late binding*

Polymorphism

- ▶ the print example is a simple example of the power of polymorphism
- ▶ the programmer can write a method using the interface of an ancestor type and the method works for objects of any descendent type
 - ▶ consider the methods of the class **java.util.Collections**

Inheritance Part 3

Liskov substitutability

- ▶ Liskov substitution principle
 - ▶ https://en.wikipedia.org/wiki/Barbara_Liskov
 - ▶ see *Substitutability* notebook for formal definition
 - ▶ subclass objects must behave the same as objects of their supertype
 - ▶ ensuring substitutability requires paying careful attention when overriding methods
- ▶ allows a programmer to write code according to the contracts of the superclass and that same code will be correct for all subclass instances

Substitutability and class invariants

- ▶ if a superclass has class invariants, then the invariants of the superclass apply to all of its descendent classes
 - ▶ invariants are public features of a class; thus, they are inherited
- ▶ a subclass that fails to support the invariants of its ancestor classes can cause correctly written code to break

Substitutability and class invariants

- ▶ for example, the **Counter** class says that the value of a counter is never negative
- ▶ therefore, the following code is technically correct

```
public class InvariantExample {  
  
    public static boolean isOdd(Counter c) {  
        return c.value() % 2 == 1;  
    }  
}
```

and will pass every possible unit test

Substitutability and class invariants

- ▶ suppose that someone creates a **DescendingCounter** class that counts down towards **Integer.MIN_VALUE** and wraps around to **Integer.MAX_VALUE**

```
public class DescendingCounter extends Counter {  
  
    public DescendingCounter() {  
        super();  
    }  
  
    public DescendingCounter(int value) {  
        super(value);  
    }  
  
    @Override  
    public void advance() {  
        this.value--;  
    }  
}
```

Substitutability and class invariants

- ▶ **isOdd** now fails when given a **DescendingCounter** instance whose value is negative

Substitutability and class invariants

- ▶ a subclass can modify the invariants inherited from its ancestors as long as the modified invariants satisfy the inherited invariants
- ▶ for example, it is possible to create a **RangedCounter** subclass of **Counter** that counts between a range of non-negative values

```
/**  
 * A counter that keeps count between two values  
 * min and max where min >= 0 and max >= min are  
 * true.  
 */  
public class RangedCounter extends Counter {  
    // implementation not shown  
}
```

The Shape class

- ▶ see *Substitutability* notebook for details
- ▶ a simple *abstract* superclass for 2D shapes

The **Shape** class

- ▶ an abstract class is a class that cannot be used to create objects directly
 - ▶ e.g., cannot make a **Shape** object directly
- ▶ the purpose of an abstract class is to serve as a superclass in an inheritance hierarchy
 - ▶ it declares all of the methods that are common to all shapes
 - ▶ it declares all of the fields that are common to all shapes
- ▶ name of an abstract class is written in italics in a UML class diagram

Shape

```
/**  
 * Superclass for 2D shapes that have an independent  
 * width and height. The width and height have a minimum  
 * value of 1 unit and a maximum value of 100 units.  
 */  
public abstract class Shape {  
  
    public static final double MIN_LENGTH = 1.0;  
    public static final double MAX_LENGTH = 100.0;  
    protected double width;  
    protected double height;  
  
    // constructors and some methods not shown
```

```
/**  
 * Returns the width of this shape. The returned  
 * width is between 1.0 and 100.0, inclusive.  
 *  
 * @return the width of this shape  
 */  
public double width() {  
    return this.width;  
}
```

```
/**  
 * Returns an upper bound on the area of this shape. Some shapes  
 * have complicated geometry and computing the exact area of such  
 * shapes is difficult. This method returns an estimate of the area  
 * of a shape where the estimate is guaranteed to be greater than or  
 * equal to the true area of the shape.  
 *  
 * <p>  
 * This method simply returns a value equal to the width times the  
 * height of the shape. Subclasses should override this method if  
 * they can provide a tighter upper bound.  
 *  
 * @return an upper bound on the area of the shape  
 */  
public double areaUpperBound() {  
    return this.width * this.height;  
}
```

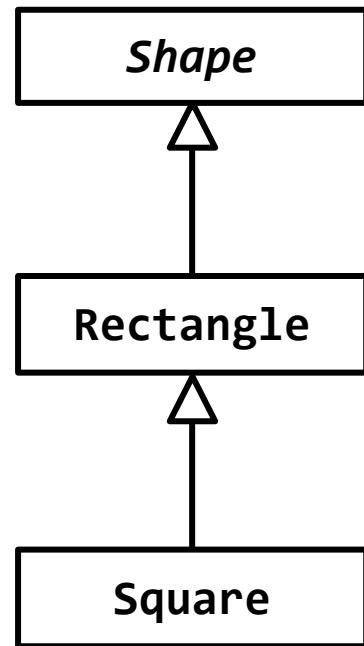
```
/**  
 * Changes both the width and height of this shape to the specified  
 * dimensions.  
 *  
 * @param width the width of the shape  
 * @param height the height of the shape  
 * @throws IllegalArgumentException if either width or height  
 * or is out of range  
 */  
public void setDimensions(double width, double height) {  
    if (width < MIN_LENGTH || height < MIN_LENGTH) {  
        throw new IllegalArgumentException("dimension too small");  
    }  
    if (width > MAX_LENGTH || height > MAX_LENGTH) {  
        throw new IllegalArgumentException("dimension too large");  
    }  
    this.width = width;  
    this.height = height;  
}
```

Using the Shape class

- ▶ a programmer can write a method that randomly changes the size of a **Shape** object
- ▶ such a method will pass every possible valid unit test

```
public class ShapeUtils {  
  
    public static void randomScale(Shape s) {  
        double w = s.width() *  
                  (1.0 + (Math.random() * 0.01 - 0.005));  
        double h = s.height() *  
                  (1.0 + (Math.random() * 0.01 - 0.005));  
  
        // make sure that setDimensions does not throw  
        if (w >= 1.0 && w <= 100.0 && h >= 1.0 && h <= 100.0) {  
            s.setDimensions(w, h);  
        }  
    }  
}
```

Extending Shape



Substitutability and parameters

- ▶ many methods place restrictions on their parameter values
 - ▶ e.g., **setDimensions** requires the width and height to be between 1 and 100
- ▶ if a subclass overrides a method, then it must accept all of the parameter values that are accepted by its ancestor classes

Substitutability and parameters

- ▶ suppose that **Square** overrides **setDimensions** so that it throws an exception if the width and height are unequal:

```
public class Square extends Rectangle {  
  
    public Square() {  
        super();  
    }  
  
    @Override  
    public void setDimensions(double width, double height) {  
        if (width != height) { // uh, oh, extra constraint  
            throw new  
                IllegalArgumentException("width != height");  
        }  
        super.setDimensions(width, height);  
    }  
}
```

Calls **Rectangle** version of
setDimensions

Substitutability and parameters

- ▶ the `randomScale` method now almost certainly fails when passed a `Square` instance because the change of width and height are chosen independently

Aside: Calling a superclass method

- ▶ when overriding a method, it is often convenient to call the superclass version of the method that is being overridden
- ▶ e.g., the **Square** version of **setDimensions** calls the **Rectangle** version of **setDimensions** to test if the width and height are in range, and to set the width and height
- ▶ to do so, use the keyword **super** followed by a period and the method name

Substitutability and parameters

- ▶ a subclass must not add or strengthen constraints on its parameter values, but it may remove or weaken constraints on its parameter values
- ▶ for example, **Rectangle** can override **setDimensions** so that out-of-range values for the width and height do not cause an exception to be thrown

```
/**  
 * Changes both the width and height of this shape to the specified  
 * dimensions. Widths or heights that are out of range are clamped  
 * to Shape.MIN_LENGTH and Shape.MAX_LENGTH.  
 *  
 * @param width the width of the shape  
 * @param height the height of the shape  
 */  
  
@Override  
public void setDimensions(double width, double height) {  
    // make sure width/height is at least MIN_LENGTH  
    width = Math.max(width, MIN_LENGTH);  
    height = Math.max(height, MIN_LENGTH);  
  
    // make sure width/height is at most MAX_LENGTH  
    width = Math.min(width, MAX_LENGTH);  
    height = Math.min(height, MAX_LENGTH);  
  
    this.width = width;  
    this.height = height;  
}
```

Substitutability and return values

- ▶ if a superclass method promises something about its return value, then a subclass that overrides the method must ensure that the overridden method returns a value that satisfies the promise of the superclass
- ▶ e.g., the **width** (and **height**) methods in **Shape** promise to return a value between 1 and 100

Using the Shape class

- ▶ a programmer can write a method that scales the dimensions of a **Shape** object so that its width is equal to 1
- ▶ we say that the method normalizes the width of the shape
 - ▶ can write similar methods to normalize the height or area of a shape
- ▶ easily done as long as the width of the shape is not zero
- ▶ such a method will pass every possible valid unit test

```
public class ShapeAnalysis {  
  
    public static void normalizeWidth(Shape s) {  
        double h = s.height() / s.width();  
        double w = 1.0;  
        s.setDimensions(w, h);  
    }  
}
```

Substitutability and return values

- ▶ suppose that a programmer creates a **VLine** class that represents a vertical line segment having zero width
- ▶ breaks an inherited class invariant, and causes **width** to return a value not between 1 and 100

```
/**  
 * A shape representing a zero-width vertical line.  
 */  
public class VLine extends Shape {  
  
    public VLine() {  
        super();  
        this.width = 0;      // uh, oh, width of zero  
    }  
  
    @Override  
    public double width() {  
        return 0.0; // uh, oh, return value not between 1-100  
    }  
}
```

Substitutability and return values

- ▶ the **normalizeWidth** method now fails when passed a **VLine** instance

Substitutability and return values

- ▶ a subclass is allowed to return a different value in an overridden method compared to its superclass if the value satisfies the contract of the superclass method
- ▶ for example, a **Triangle** class could return the exact area of the triangle in the **areaUpperBound** method

```
public class Triangle extends Shape {  
  
    public Triangle() {  
        super();  
    }  
  
    /**  
     * Returns the true area of this triangle.  
     *  
     * @return the area of this triangle  
     */  
    @Override  
    public double areaUpperBound() {  
        return 0.5 * this.width * this.height;  
    }  
}
```

Substitutability and mutator methods

- ▶ if a superclass method promises to change the state of an object in a particular way, then a subclass that overrides the method must ensure that the overridden method changes the state of an object in a way that satisfies the promise of the superclass

Substitutability and mutator methods

- ▶ suppose the implementer of **Square** "fixes" their version of **setDimensions** as shown on the next slide so that it no longer imposes the constraint that the specified width and height must be equal
- ▶ describe a situation where code written using the contracts from the **Shape** class fails when passed a **Square** object

```
public class Square extends Rectangle {  
  
    public Square() {  
        super();  
    }  
  
    @Override  
    public void setDimensions(double width, double height) {  
        // ignore height parameter  
        super.setDimensions(width, width);  
    }  
}
```

Substitutability and mutator methods

- ▶ suppose the implementer of **Square** "fixes" their version of **setDimensions** as shown on the next slide so that it no longer imposes the constraint that the specified width and height must be equal
- ▶ describe a situation where code written using the contracts from the **Shape** class fails when passed a **Square** object

```
public class Square extends Rectangle {  
  
    public Square() {  
        super();  
    }  
  
    public void setLength(double length) {  
        super.setDimensions(length, length);  
    }  
  
    @Override  
    public void setDimensions(double width, double height) {  
        // do nothing  
    }  
}
```

Java Generics

A Stack of int

- ▶ recall our **IntStack** class

```
import java.util.Arrays;
import java.util.EmptyStackException;

public class IntStack {
    // the initial capacity of the stack
    private static final int DEFAULT_CAPACITY = 16;

    // the array that stores the stack
    private int[] stack;

    // the index of the top of the stack (equal to -1 for an empty stack)
    private int topIndex;

    public void push(int element) { // implementation not shown }

    public int pop() { // implementation not shown }

}
```

A Stack of String

- ▶ suppose we wanted a stack of strings
- ▶ before the introduction of generics into the Java language one solution would have been to create a new class
- ▶ **StringStack**

```
import java.util.Arrays;
import java.util.EmptyStackException;

public class StringStack {
    // the initial capacity of the stack
    private static final int DEFAULT_CAPACITY = 16;

    // the array that stores the stack
    private String[] stack;

    // the index of the top of the stack (equal to -1 for an empty stack)
    private int topIndex;

    public void push(String element) { // implementation not shown }

    public String pop() { // implementation not shown }

}
```

A Stack of String

- ▶ almost nothing changes in the implementation of **StringStack** compared to **IntStack**
- ▶ consider the **pop** method

```
/**  
 * Pop and return the top element of the stack.  
 *  
 * @return the top element of the stack  
 * @throws EmptyStackException if the stack is empty  
 */  
public String pop() {  
    // is the stack empty?  
    if (this.topIndex == -1) {  
        throw new EmptyStackException();  
    }  
    // get the element at the top of the stack  
    String element = this.stack[this.topIndex];  
  
    // adjust the top of stack index  
    this.topIndex--;  
  
    // return the element that was on the top of the stack  
    return element;  
}
```

A Stack of String

- ▶ the only things that change are related to type declarations
 - ▶ we substitute **String** for **int** in a few places
- ▶ wouldn't it be nice if we could get the compiler to perform this substitution for us?
- ▶ this is exactly what generics in Java are for

Java pre-generics

- ▶ before Java generics were introduced to the language, generic collections were implemented as collections of **Object** references
- ▶ for example, we can create a class representing a stack of objects

```
public class ListObjectStack {  
  
    private ArrayList<Object> stack;  
  
    public ListObjectStack() {  
        this.stack = new ArrayList<>();  
    }  
  
    public void push(Object elem) {  
        this.stack.add(elem);  
    }  
  
    public Object pop() {  
        if (this.isEmpty()) {  
            throw new RuntimeException("popped an empty stack");  
        }  
        Object elem = this.stack.remove(this.size() - 1);  
        return elem;  
    }  
}
```

Using a stack of objects

- ▶ using such a stack can be inconvenient because the declared type of the elements is **Object**
- ▶ popping an element from the stack usually requires a cast before the element can be used

```
ListObjectStack stringStack = new ListObjectStack();
stringStack.push("hello");
stringStack.push("konnichiwa");
stringStack.push("hallo");

// cast needed because pop returns an Object reference
String s = (String) stringStack.pop();

// now the element can be used as a String
System.out.println(s.toUpperCase());
```

Lack of type safety

- ▶ a more serious problem is that any reference can be pushed onto a stack of objects
- ▶ you can create a stack that is supposed to hold only strings, but there is nothing to prevent someone from pushing some other type of element onto the stack
 - ▶ the compiler cannot help you here so we say that such a stack lacks type safety

```
import java.util.Date;

ListObjectStack stringStack = new ListObjectStack();
stringStack.push("hello");
stringStack.push("konnichiwa");
stringStack.push("hallo");

stringStack.push(1);    // works, 1 is autoboxed to an Integer object

stringStack.push(new ListObjectStack());          // works

stringStack.push(new Date());                     // works
```

Lack of type safety

- ▶ the fact that any type of element can be pushed onto a stack makes it impossible to write type safe methods that operate on collections of objects
- ▶ such methods and programs that use such methods will *compile cleanly* but fail at runtime

```
public class Sum {  
  
    public static int sumAndClear(List<Object> intStack) {  
        int sum = 0;  
        while (intStack.size() > 0) {  
            Object elem = intStack.pop();  
            // cast needed because pop returns an Object  
            // throws an exception if elem is not an Integer  
            sum += (Integer) elem;  
        }  
        return sum;  
    }  
}
```

Java Generics

- ▶ generics in Java allow us to parameterize a class, interface, or method over types
- ▶ instead of creating separate classes **IntStack**, **StringStack**, **DateStack**, etc. we create a single class and specify the type parameter:
 - ▶ **Stack<Integer>** instead of **IntStack**
 - ▶ **Stack<String>** instead of **StringStack**
 - ▶ **Stack<Date>** instead of **DateStack**
- ▶ such generic classes are type safe

Declaring a Generic Class

- ▶ declare a generic class using the following syntax:

```
class ClassName<T1, T2, ..., Tn>
```

- ▶ the angled brackets denote the type parameter section that specifies the *type parameters* (also called *type variables*)
- ▶ the type parameters can be any non-primitive type

Type Parameter Naming Conventions

- ▶ type parameters are single, uppercase letters
- ▶ common type parameter names:
 - ▶ **E** – Element
 - ▶ **K** – Key
 - ▶ **N** – Number
 - ▶ **T** – Type
 - ▶ **V** – Value

A Simple Generic Class Example

- ▶ the fact that a Java method can return only a single value is sometimes inconvenient
 - ▶ for example, it would be nice if a method that found the maximum element in a list would return the maximum element and the location of the element in the list
 - ▶ in Python, you can return a tuple of values
- ▶ you can easily make a simple generic class that represents a pair of values
 - ▶ such a class has no constructors or methods

```
public class Pair<T1, T2> {  
    public T1 first;  
    public T2 second;  
}
```

```
public class ListUtil {  
  
    public static Pair<Integer, Integer> max(List<Integer> t) {  
        if (t.isEmpty) {  
            throw new IllegalArgumentException();  
        }  
        Pair<Integer, Integer> result;  
        result.first = 0;  
        result.second = t.get(0);  
        for (int i = 1; i < t.size(); i++) {  
            int elem = t.get(i);  
            if (elem > result.second) {  
                result.first = i;  
                result.second = elem;  
            }  
        }  
        return result;  
    }  
}
```

A Generic Stack Class

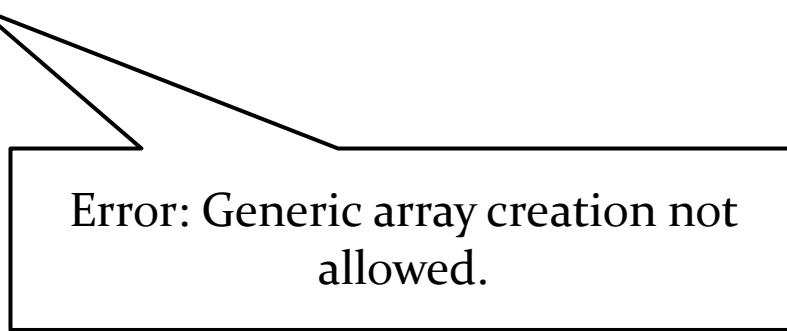
```
import java.util.Arrays;
import java.util.EmptyStackException;

public class ArrayStack<E> {
    // the initial capacity of the stack
    private static final int DEFAULT_CAPACITY = 16;

    // the array that stores the stack
    private E[] stack;

    // the index of the top of the stack
    private int topIndex;
```

```
/**  
 * Create an empty stack.  
 */  
  
public ArrayStack() {  
    this.stack = new E[Stack.DEFAULT_CAPACITY];  
    this.topIndex = -1;  
}
```



Error: Generic array creation not
allowed.

No Arrays of Generic Type

- ▶ the highlighted line on the previous slide causes a compilation error
 - ▶ in Java you cannot create an array of generic type
- ▶ a solution is to make an array of **Object**
 - ▶ requires **pop** to cast the element to the generic type

A Generic Stack Class

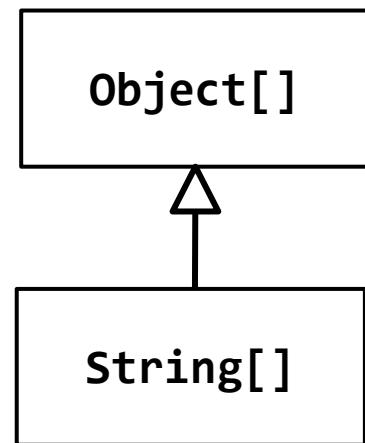
```
import java.util.Arrays;  
import java.util.EmptyStackException;  
  
public class ArrayStack<E> {  
    // the initial capacity of the stack  
    private static final int DEFAULT_CAPACITY = 16;  
  
    // the array that stores the stack  
    private Object[] stack;  
  
    // the index of the top of the stack  
    private int topIndex;
```

Change from E[] to Object[]

```
/**  
 * Create an empty stack.  
 */  
public ArrayStack() {  
    this.stack = new Object[Stack.DEFAULT_CAPACITY];  
    this.topIndex = -1;  
}
```

Arrays are Covariant

- ▶ arrays in Java are said to be *covariant*
 - ▶ if **Sub** is a subtype of **Super** then the array type **Sub[]** is a subtype of the array type **Super[]**



-
- ▶ does the following compile?

```
Object[] objs = new Integer[1];
```

- A. yes
- B. no

-
- ▶ does the following compile?

```
Object[] objs = new Integer[1];  
objs[0] = "Hello";
```

- A. yes
- B. no

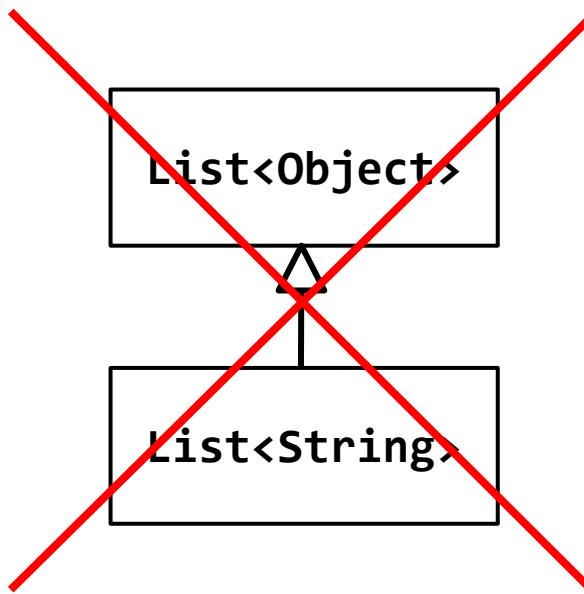
-
- ▶ does the following run cleanly?

```
Object[] objs = new Integer[1];  
objs[0] = "Hello";
```

- A. yes
- B. no

Generics are not Covariant

- ▶ generics are *invariant*
 - ▶ for any two distinct types **T1** and **T2**, **List<T1>** is neither a subtype nor a supertype of **List<T2>**



-
- ▶ does the following compile?

```
List<Object> objs = new ArrayList<Integer>();
```

- A. yes
- B. no

A Generic Stack Class

- ▶ the methods **push** and **pop** can be implemented with only very minor changes compared to **IntStack**

```
public void push(E element) {  
    // is there capacity for one more element?  
    if (this.topIndex < this.stack.length - 1) {  
        // increment the top of stack index and insert the element  
        this.topIndex++;  
        this.stack[this.topIndex] = element;  
    } else {  
        Object[] newStack = Arrays.copyOf(this.stack,  
                                         this.stack.length * 2);  
        // refer to the new array and push the element onto the stack  
        this.stack = newStack;  
        this.push(element);  
    }  
}
```

```
public E pop() {  
    // is the stack empty?  
    if (this.topIndex == -1) {  
        throw new EmptyStackException();  
    }  
    // get the element at the top of the stack  
    E element = (E) this.stack[this.topIndex];  
  
    // adjust the top of stack index  
    this.topIndex--;  
  
    // return the element that was on the top of the stack  
    return element;  
}
```

A Generic Stack Interface

- ▶ converting our stack-of-string interface to a generic stack interface is very easy to do
- ▶ just add the generic type variable after the interface name and change **String** to the generic variable name
- ▶ our original interface is shown on the next slide:

```
package ca.queensu.cs.cisc124.notes.interfaces;

public interface Stack {

    public int size();

    default boolean isEmpty() {
        return this.size() == 0;
    }

    public void push(String elem);

    public String pop();
}
```

```
package ca.queensu.cs.cisc124.notes.generics.basics;
```

```
public interface Stack<E> {
```

Generic type variable

```
    public int size();
```

```
    default boolean isEmpty() {
```

```
        return this.size() == 0;
```

```
}
```

```
    public void push(E elem);
```

Change **String** to **E**

```
    public E pop();
```

Change **String** to **E**

```
}
```

ArrayStack implements Stack

- ▶ finally, we can change the declaration of the **ArrayStack** class to say that it implements our generic **Stack** interface

ArrayStack implements Stack

```
import java.util.Arrays;  
import java.util.EmptyStackException;  
  
public class ArrayStack<E> implements Stack<E> {  
    // the initial capacity of the stack  
    private static final int DEFAULT_INITIAL_CAPACITY = 10;  
    // Add implements Stack<E>  
  
    // the array that stores the stack  
    private Object[] stack;  
  
    // the index of the top of the stack  
    private int topIndex;
```

Generic queues

Queue

- ▶ example of a queue



Front and back of a queue



Queue Operations

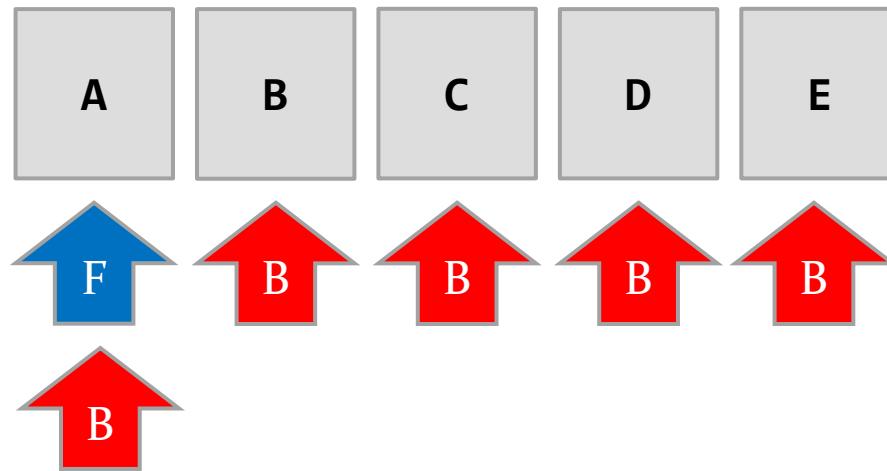
- ▶ classically, queues only support two operations
 - 1. enqueue
 - ▶ add to the back of the queue
 - 2. dequeue
 - ▶ remove from the front of the queue
 - ▶ error to dequeue an empty queue

Queue Optional Operations

- ▶ optional operations
 - 1. `size`
 - ▶ number of elements in the queue
 - 2. `isEmpty`
 - ▶ is the queue empty?
 - 3. `peek`
 - ▶ get the front element (without removing it)
 - 4. `contains`
 - ▶ find the position of a specified element in the queue

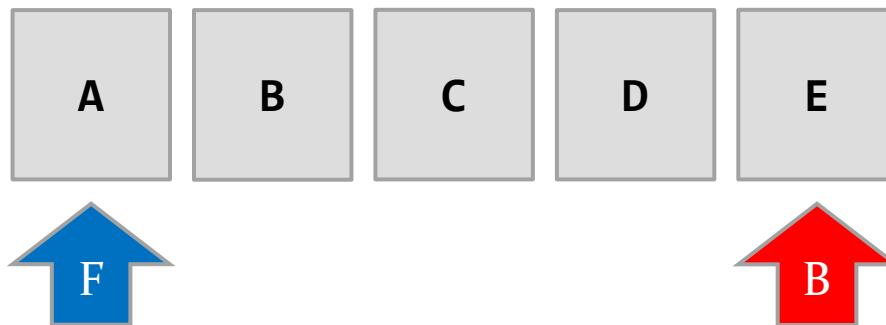
Enqueue

1. **q.enqueue("A")**
2. **q.enqueue("B")**
3. **q.enqueue("C")**
4. **q.enqueue("D")**
5. **q.enqueue("E")**



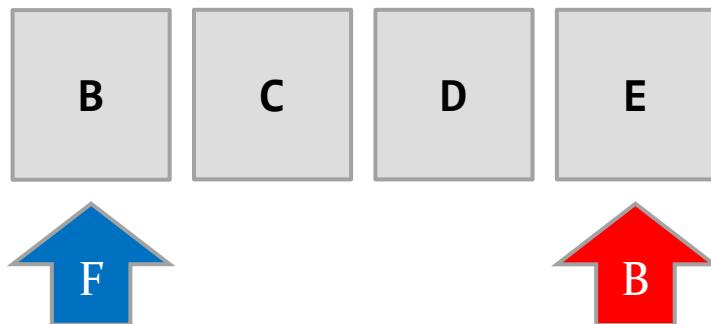
Dequeue

1. **String s = q.dequeue()**



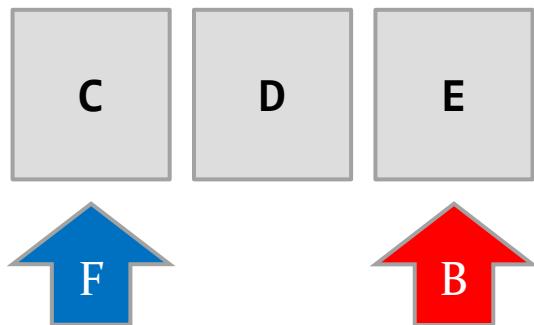
Dequeue

1. **String s = q.dequeue()**
2. **s = q.dequeue()**



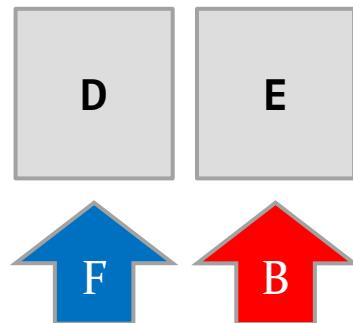
Dequeue

1. **String s = q.dequeue()**
2. **s = q.dequeue()**
3. **s = q.dequeue()**



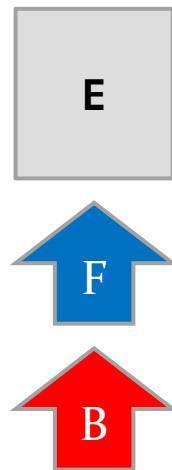
Dequeue

1. **String s = q.dequeue()**
2. **s = q.dequeue()**
3. **s = q.dequeue()**
4. **s = q.dequeue()**



Dequeue

1. **String s = q.dequeue()**
2. **s = q.dequeue()**
3. **s = q.dequeue()**
4. **s = q.dequeue()**
5. **s = q.dequeue()**



FIFO

- ▶ queue is a First-In-First-Out (FIFO) data structure
 - ▶ the first element enqueued in the queue is the first element that can be accessed from the queue

Applications

- ▶ queues are used widely in computer science and computer engineering
 - ▶ online customer queues
 - ▶ web servers use queues to service requests
 - ▶ operating systems use queues to distribute resources
 - ▶ printer, disk, CPU queues
 - ▶ network devices use queues to determine which packets to service
 - ▶ breadth-first search
 - ▶ https://en.wikipedia.org/wiki/Queueing_theory

Queue interface

```
public interface Queue<E> {  
    public int size();  
  
    default boolean isEmpty() {  
        return this.size() == 0;  
    }  
  
    public void enqueue(E elem);  
  
    public E dequeue();  
  
    public E front(); // returns front element  
  
    public E back(); // returns back element  
}
```

Implementation with List

- ▶ a queue implementation using a list is similar to that for a stack except that **dequeue** removes the element at the front of the list

```
import java.util.List;
import java.util.ArrayList;

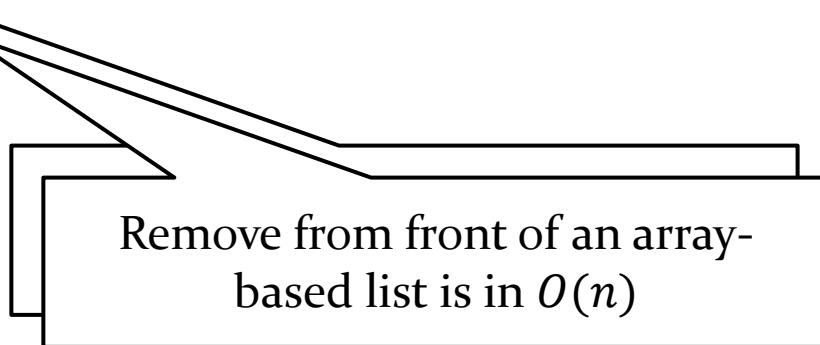
public class ListQueue<E> implements Queue<E> {
    private List<E> q;

    public ListQueue() {
        this.q = new ArrayList<>();
    }

    @Override
    public int size() {
        return this.q.size();
    }

    @Override
    public void enqueue(E elem) {
        this.q.add(elem);
    }
}
```

```
private void throwIfEmpty() {  
    if (this.isEmpty()) {  
        throw new RuntimeException("empty queue");  
    }  
}  
  
@Override  
public E dequeue() {  
    this.throwIfEmpty();  
    return this.q.remove(0);  
}
```



Remove from front of an array-based list is in $O(n)$

```
@Override
public E front() {
    this.throwIfEmpty();
    return this.q.get(0);
}

@Override
public E back() {
    this.throwIfEmpty();
    return this.q.get(this.q.size() - 1);
}

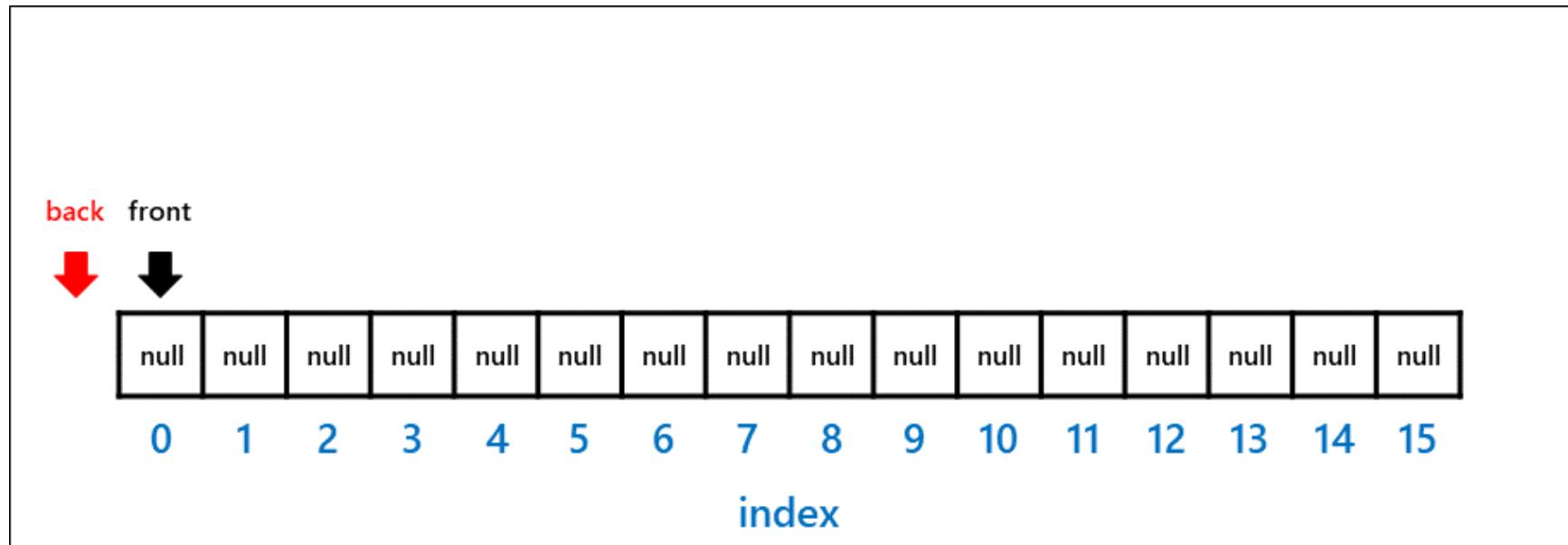
@Override
public String toString() {
    return this.q.toString();
}
}
```

Implementation with an array

- ▶ the fact that removal and additions occur at different ends of the queue make an array-based implementation somewhat tricky
- ▶ we use 4 fields:

Field name	Purpose
arr	array for the elements of the queue
size	number of elements in the queue
front	array index of element currently at the front of the queue
back	array index of element currently at the back of the queue

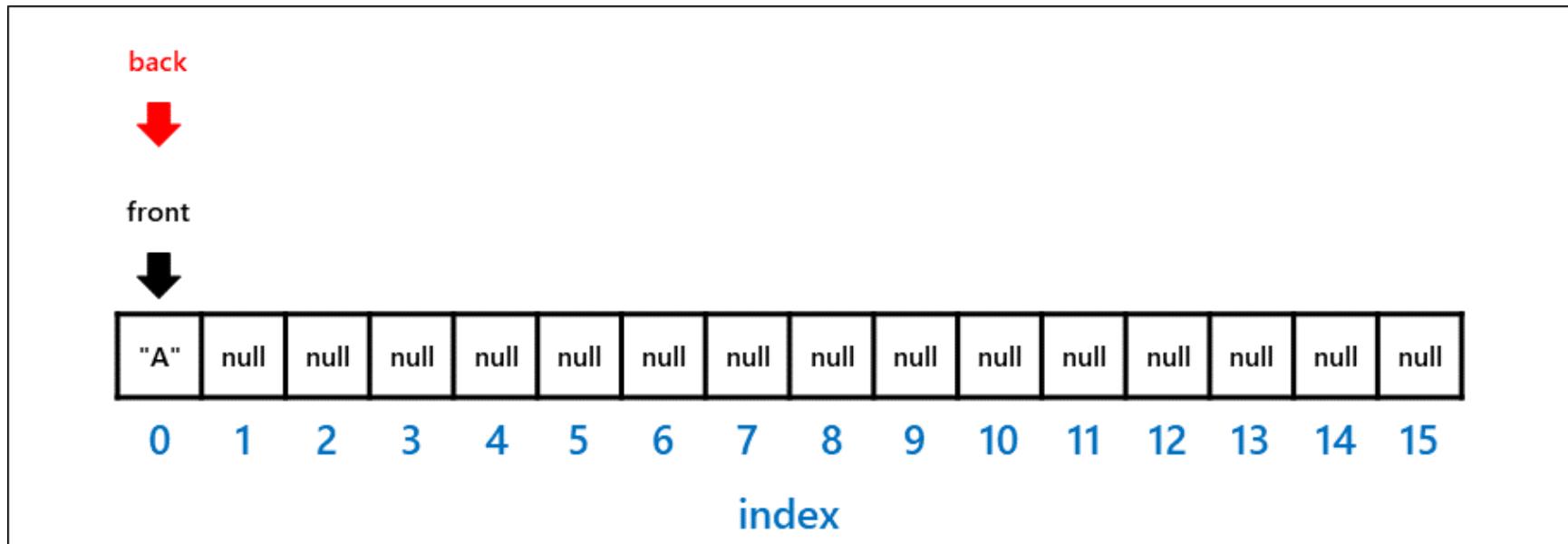
New empty queue



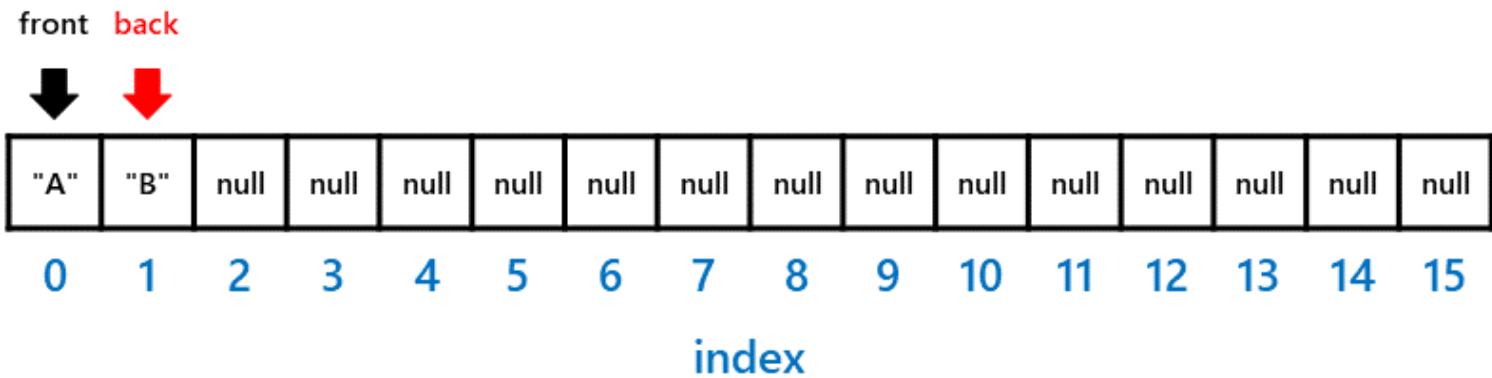
enqueue algorithm

- ▶ enqueueing an element is similar to pushing an element onto the top of a stack
1. if **arr[length] == size**, then make a new array with greater capacity and copy the existing elements into the new array
 2. move **back** one position to the right
 3. set the element at **arr[back]** to the enqueued element

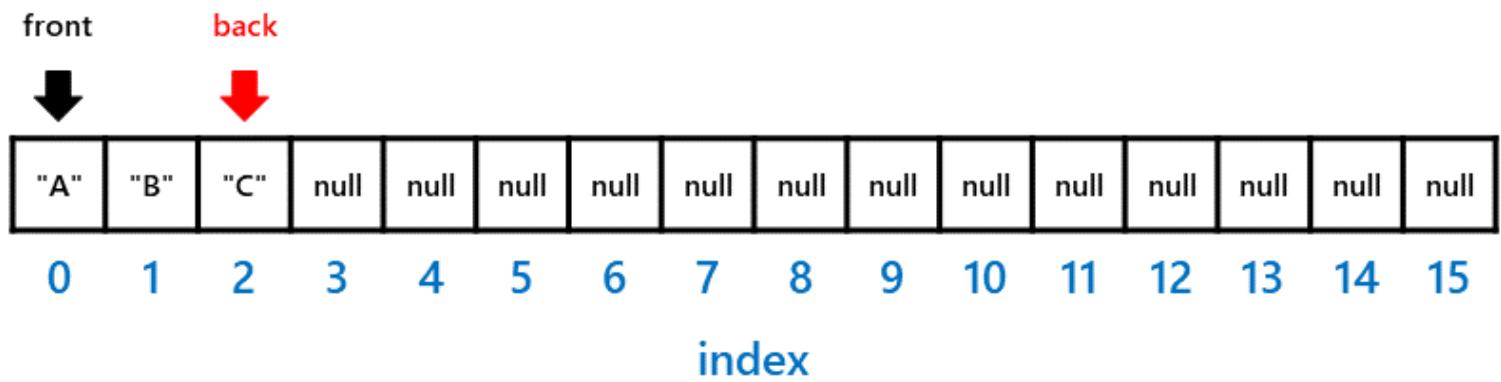
After one enqueue



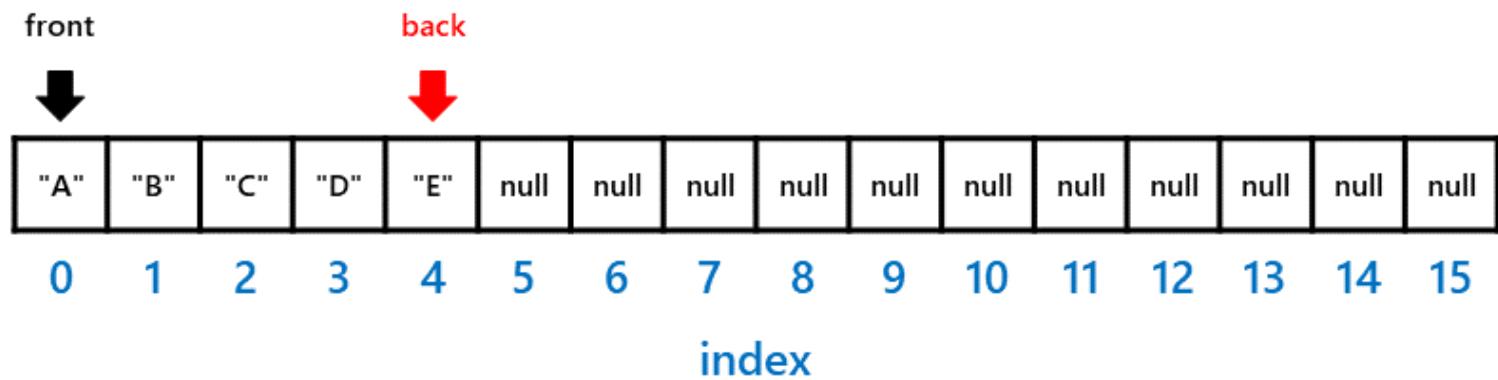
After another enqueue



After another enqueue



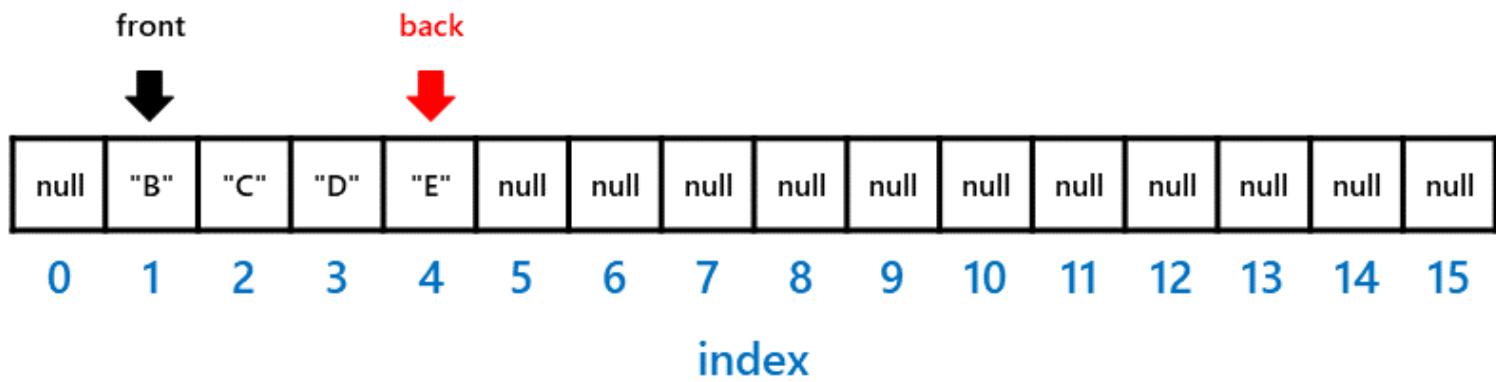
After five enqueue operations



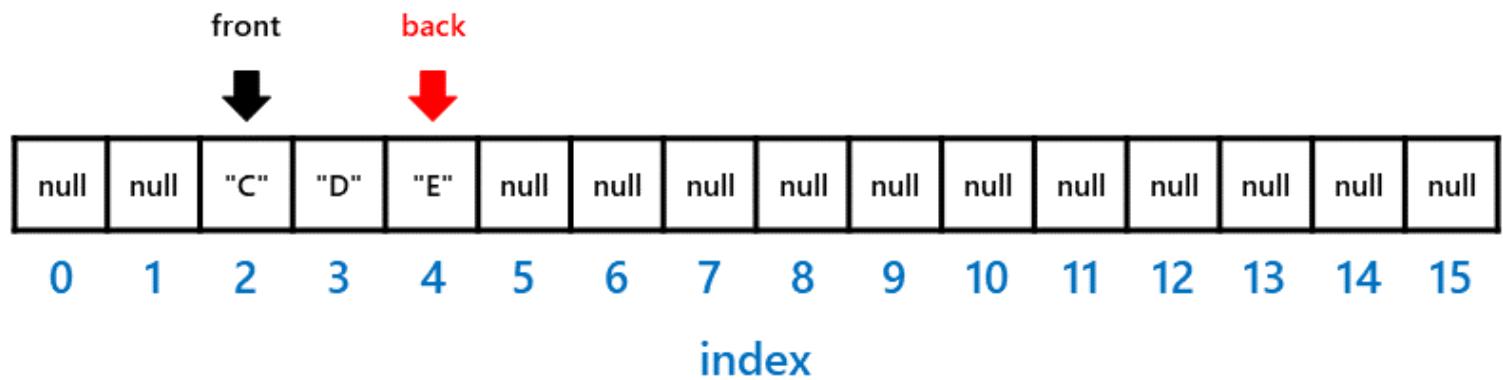
dequeue algorithm

1. **elem = arr[front]**
2. **arr[front] = null**
3. **front** moves one position to the right
4. **return elem**

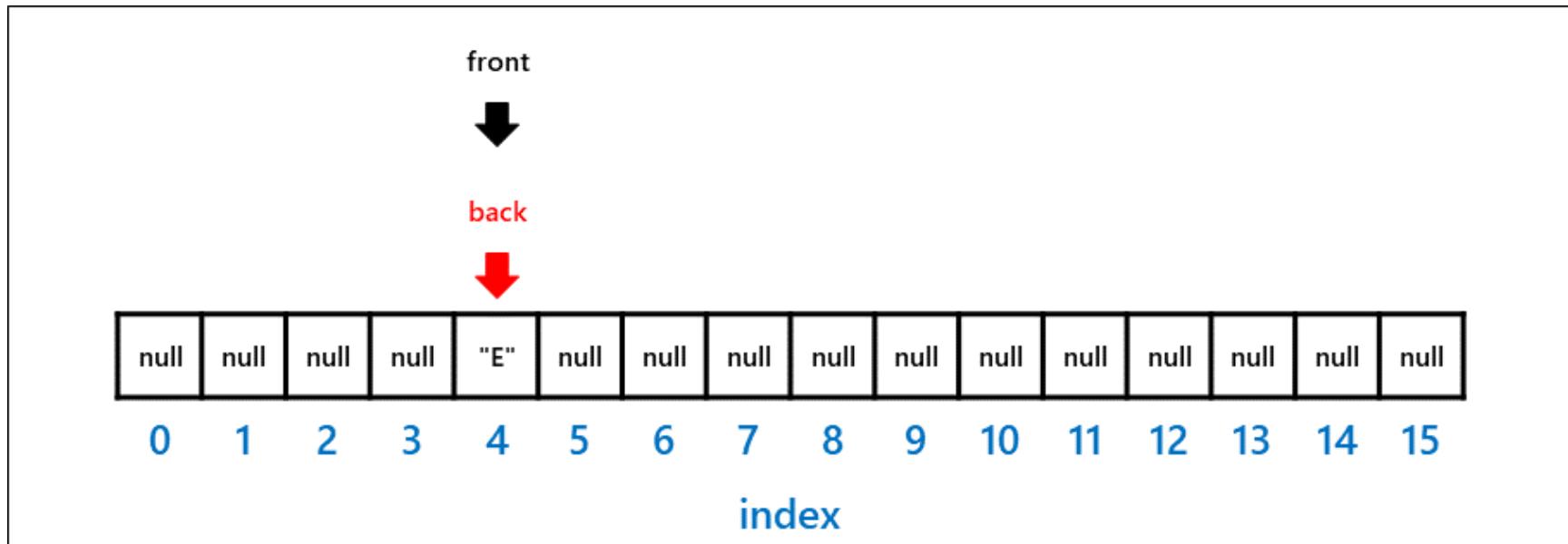
After one dequeue



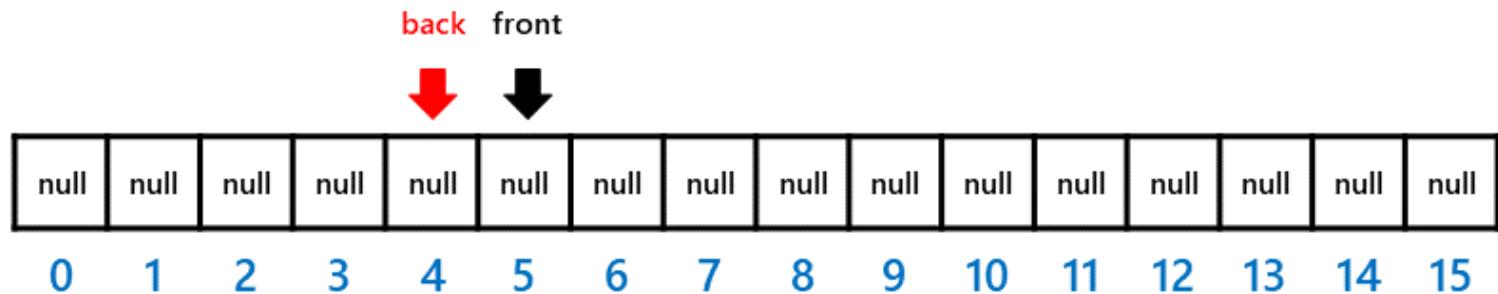
After two dequeue operations



After four dequeue operations

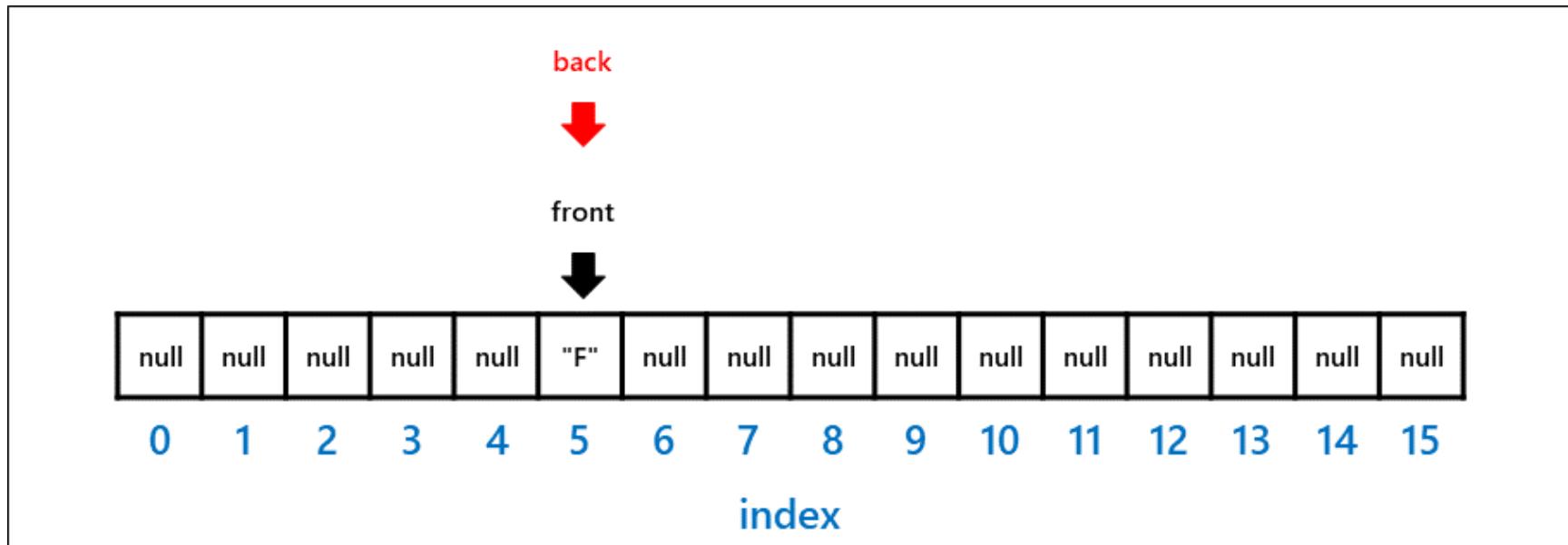


After five dequeue operations

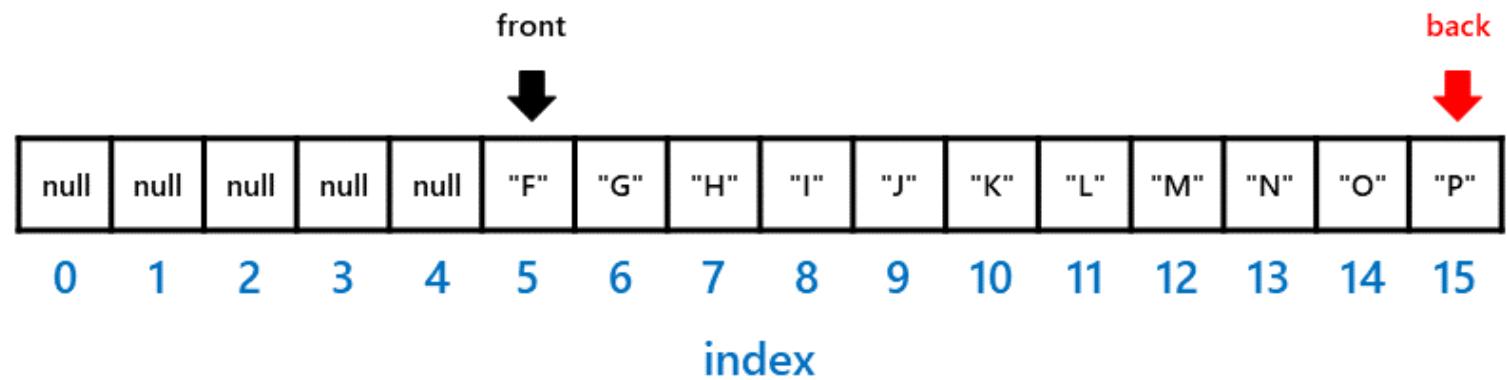


back is one less than **front** in this empty queue, but this is not a reliable indicator of an empty queue

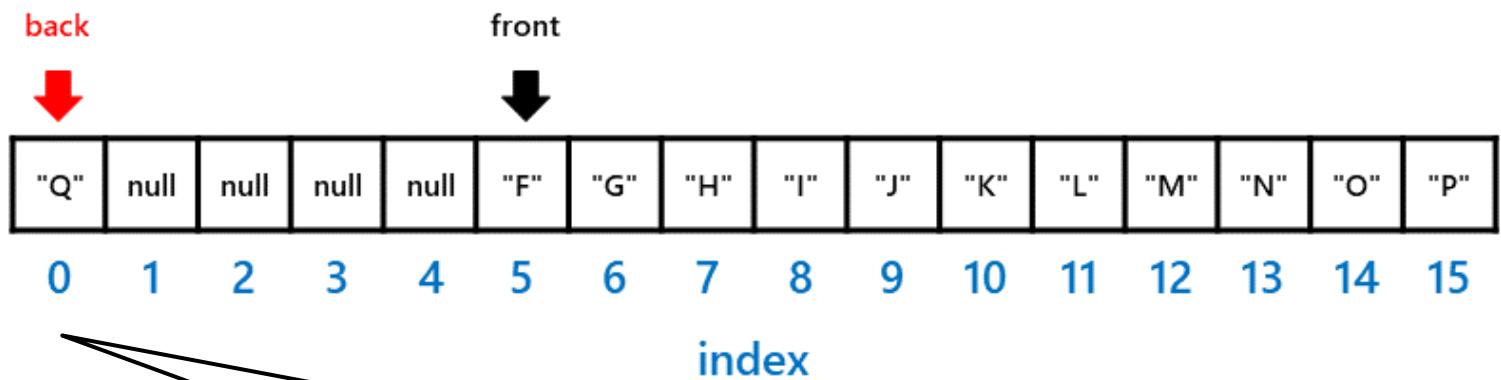
After one enqueue operation



After ten more enqueue operations

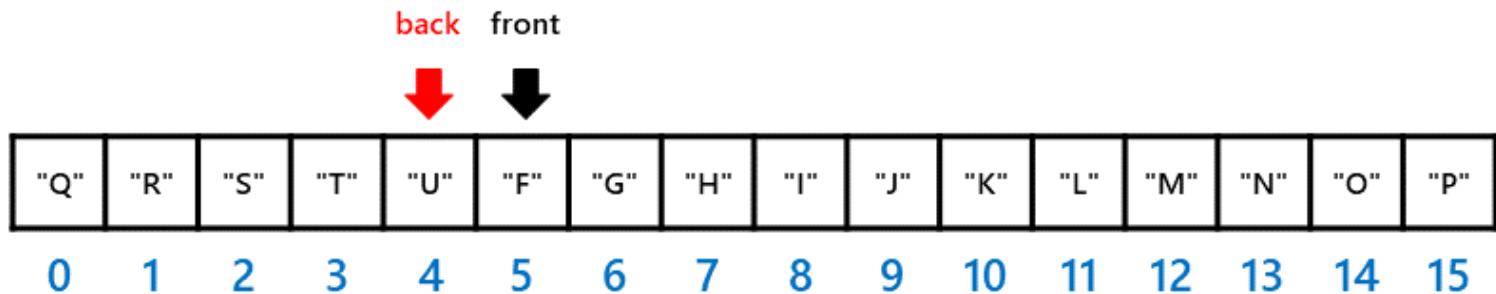


After one more enqueue operation



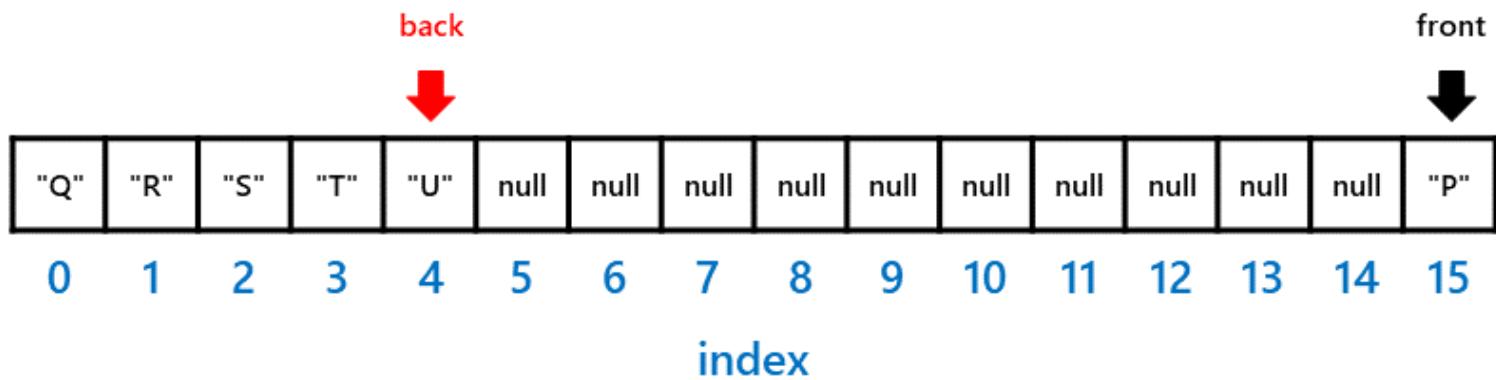
back wraps around to the front of the array
back can be less than **front** in a non-empty queue

Example of a full queue

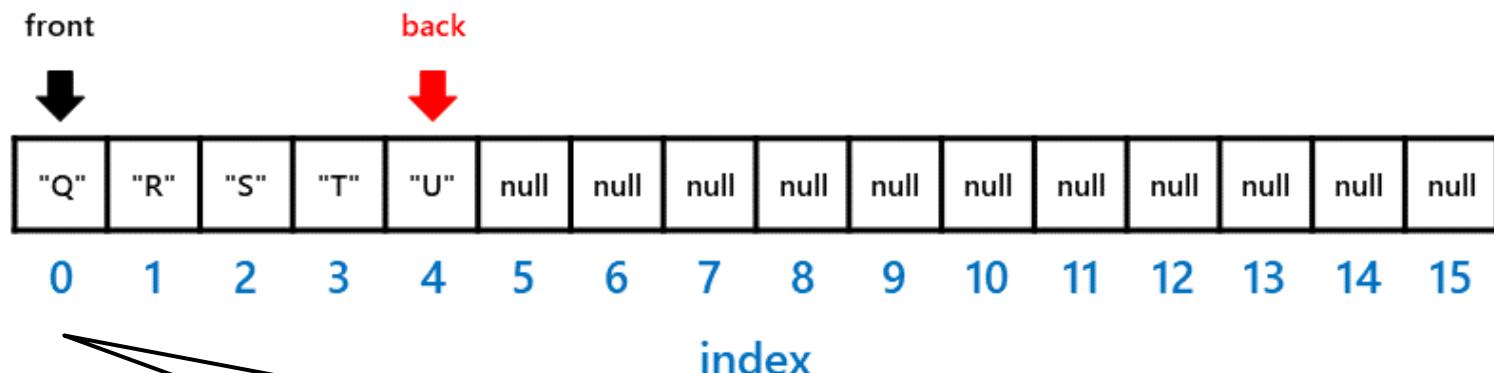


back is one less than **front** in this full queue (same as in the previous empty queue example)

After eleven more dequeue operations

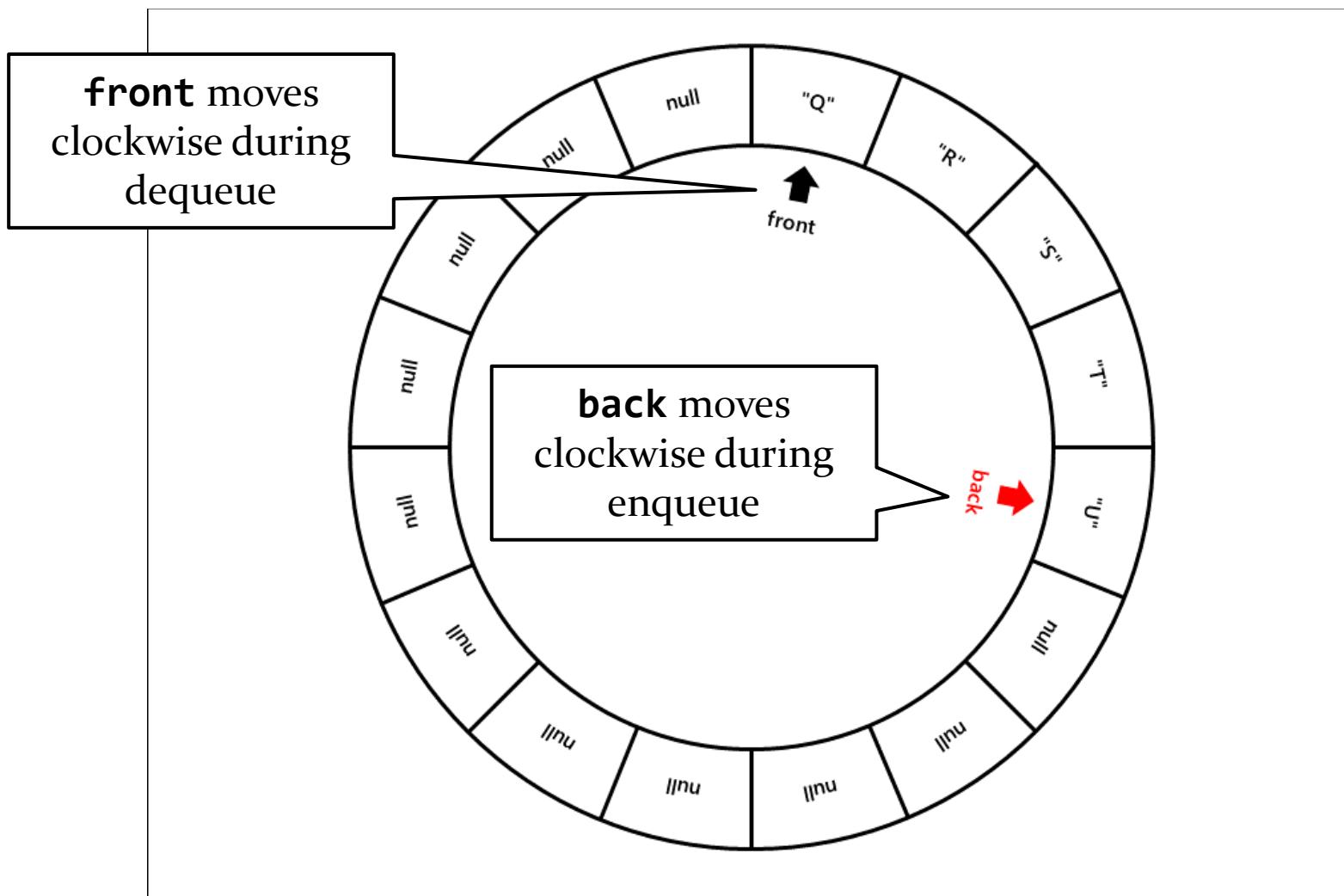


After one more dequeue operation



front wraps around to the front of the array

Queue as a circular array



```
public class ArrayQueue<E> implements Queue<E> {  
  
    // the default capacity of the queue  
    private static final int DEFAULT_CAPACITY = 16;  
  
    // the array that stores the queue  
    private Object[] arr;  
  
    // the index of the element at the front of the queue  
    private int front;  
  
    // the index of the element at the back of the queue  
    private int back;  
  
    // the number of elements in the queue  
    private int size;
```

```
public ArrayQueue() {  
    this.arr = new Object[ArrayQueue.DEFAULT_CAPACITY];  
    this.front = 0;  
    this.back = -1;  
    this.size = 0;  
}  
  
public ArrayQueue(ArrayQueue<E> other) {  
    this.arr = Arrays.copyOf(other.arr, other.arr.length);  
    this.front = other.front;  
    this.back = other.back;  
    this.size = other.size;  
}
```

```
@Override  
public E front() {  
    if (this.isEmpty()) {  
        throw new RuntimeException("no front element");  
    }  
    return (E) this.arr[this.front];  
}
```

```
@Override  
public E back() {  
    if (this.isEmpty()) {  
        throw new RuntimeException("no back element");  
    }  
    return (E) this.arr[this.back];  
}
```

```
@Override
public E dequeue() {
    if (this.isEmpty()) {
        throw new RuntimeException("dequeued an empty queue");
    }
    Object elem = this.arr[this.front];

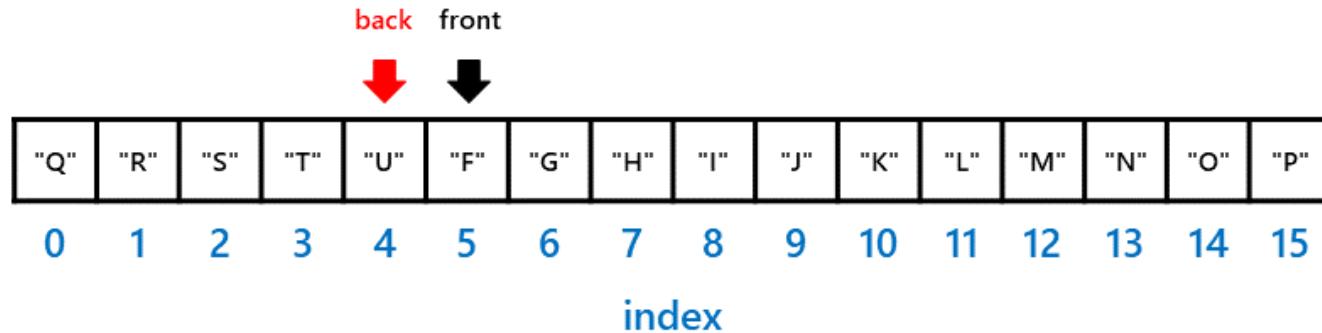
    // null out the value stored in the array
    this.arr[this.front] = null;
    this.front = (this.front + 1) % this.arr.length;
    this.size--;
    return (E) elem;
}
```

Resizing the array

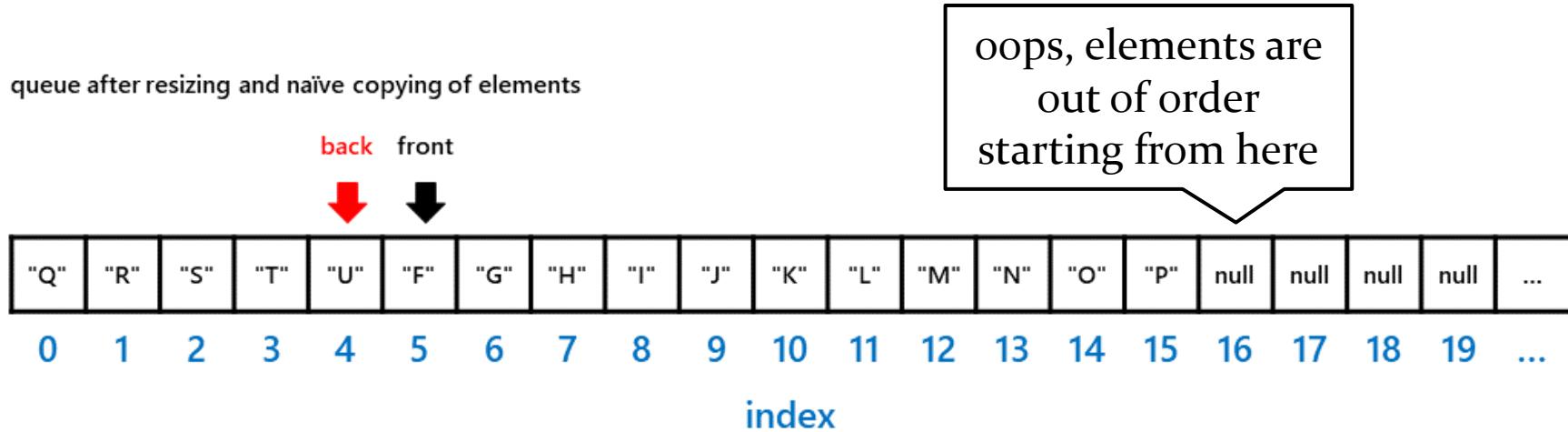
- ▶ resizing the internal array when it reaches its capacity is slightly more complicated than for a stack because the order of the elements is not necessarily from the beginning of the array to the end

Naïve array resizing

full queue before resizing



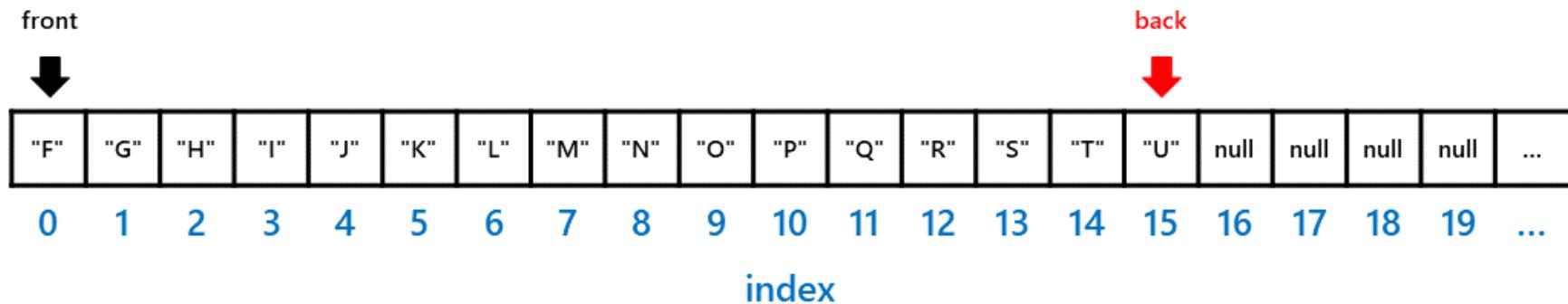
queue after resizing and naïve copying of elements



Correct array resizing

- one way to correctly copy the elements from the old array is to put the front element of the queue at index **0** of the new array

queue after resizing and correct copying of elements



```
@Override
public void enqueue(E elem) {
    if (this.size == this.arr.length) {
        Object[] tmp = new Object[this.arr.length * 2];
        for (int i = 0; i < this.size; i++) {
            tmp[i] = this.arr[(this.front + i) %
                               this.arr.length];
        }
        this.arr = tmp;
        this.front = 0;
        this.back = this.size - 1;
    }
    this.back = (this.back + 1) % this.arr.length;
    this.arr[this.back] = elem;
    this.size++;
}
```

Exercises for the student

- ▶ implement the following copy constructor without using a temporary collection

```
public ArrayQueue(Queue<E> other)
```

- ▶ implement **equals** where a **Queue<E>** instance can be equal to an **ArrayQueue<E>** instance

Generic queues with linked nodes

Stack using linked nodes

- ▶ review Lecture 22 Stacks 2

Queues using linked nodes

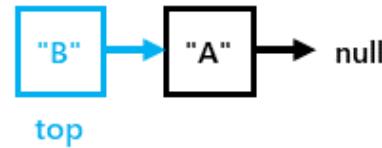
- ▶ the main difference between a linked-node based stack and queue is where nodes are added
 - ▶ in a stack, nodes are added at the front of the chain of nodes
 - ▶ in a queue, nodes are added at the end of the chain of nodes

Stack (nodes add at front)

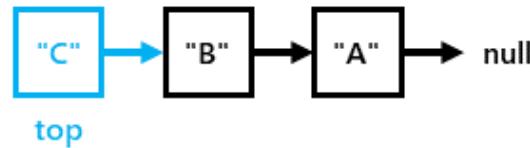
t.push("A")



t.push("B")



t.push("C")

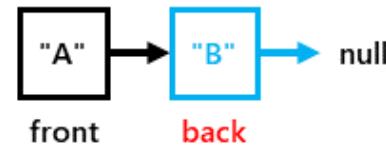


Queue (nodes at at back)

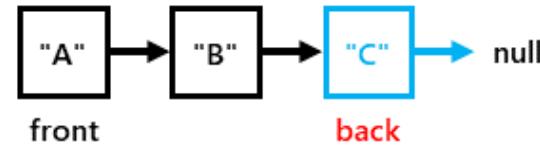
q.enqueue("A")



q.enqueue("B")



q.enqueue("C")



```
public class LinkedQueue<E> implements Queue<E> {
    // the number of elements currently in the queue
    private int size;

    // the nodes containing the front and back elements of the queue
    private Node<E> front;
    private Node<E> back;

    // static nested class representing nodes of the linked list
    private class Node<E> {

        // the element stored in the node
        E elem;

        // the link to the next node in the sequence
        Node<E> next;

        Node(E elem, Node<E> next) {
            this.elem = elem;
            this.next = next;
        }
    }
}
```

```
public LinkedQueue() {
    this.size = 0;
    this.front = null;
    this.back = null;
}

@Override
public E front() {
    if (this.isEmpty()) {
        throw new RuntimeException("no front element");
    }
    return this.front.elem;
}

@Override
public E back() {
    if (this.isEmpty()) {
        throw new RuntimeException("no back element");
    }
    return this.back.elem;
}
```

```
@Override
public void enqueue(E elem) {
    Node<E> n = new Node<>(elem, null);

    if (this.isEmpty()) {
        // special case: modify this.front
        this.front = n;
    }
    else if (this.size() == 1) {
        // special case: modify this.front.next
        this.front.next = n;
    }
    else {
        this.back.next = n;
    }
    this.back = n;
    this.size++;
}
```

```
@Override
public E dequeue() {
    if (this.isEmpty()) {
        throw new RuntimeException("dequeued an empty queue");
    }
    E elem = this.front.elem;
    this.front = this.front.next;
    this.size--;
    if (this.isEmpty()) {
        // special case; need to update this.back
        this.back = null;
    }
    return elem;
}
```

Complexity

- ▶ complexity of **enqueue** is in $O(1)$
 - ▶ worst-case complexity is better than for an array-based queue
 - ▶ average-case complexity is the same as for an array-based queue, but...
 - ▶ average-case runtime is worse than for an array-based queue
- ▶ complexity of **dequeue** is in $O(1)$
 - ▶ worst-case complexity is the same as for an array-based queue
 - ▶ average-case complexity and runtime is similar to an array-based queue

Generic methods

Generic methods

- ▶ a generic method is a method that has its own type variables
- ▶ may appear in a generic class, but for simplicity, we will define generic methods in a non-generic class
- ▶ the type variables for each method are independent of one another
- ▶ i.e., the scope of the type variables is confined to the method

Generic methods

- ▶ generic methods often appear in utility classes that operate on generic types
 - ▶ `java.util.Arrays`
 - ▶ `java.util.Collections`
 - ▶ `java.util.Objects`

Defining a generic method

- ▶ a generic method looks exactly like any other method except that the type variables appear inside < > before the return type of the method
- ▶ let's create a utility class that operates on stacks
 - ▶ we'll start with a method that clears a stack (removes all of the elements)
 - ▶ the element type of the stack is unimportant in this method because we are not using the elements in this method

```
public class Stacks { // class itself is not generic
                      // no type variable after class name

// method is generic, type variable appears before return type
public static <T> void clear(Stack<T> s) {
    while (s.size() > 0) {
        s.pop();
    }
}
```

Defining a generic method

- ▶ the **pushAll** methods push all of the elements in a collection or an array onto a stack
 - ▶ the methods do not remove the elements from the collection or array
- ▶ type safety matters in these methods
 - ▶ the element type of the collection/array must match (?) the element type of the stack

```
public class Stacks { // class itself is not generic
    // no type variable after class name

    public static <T> void clear(Stack<T> s) { // not shown }

    // method is generic, type variable appears before return type
    public static <T> void pushAll(Collection<T> src, Stack<T> dest) {
        for (T elem : src) {
            dest.push(elem);
        }
    }

}
```

// method is generic, type variable appears before return type

Element types match

Defining a generic method

- ▶ the **popAll** methods pop all of the elements off a stack and add them to a collection or array
- ▶ the methods do not remove the elements from the collection or array
- ▶ type safety matters in these methods
 - ▶ the element type of the stack must match (?) the element type of the collection/array

```
public class Stacks { // class itself is not generic
    // no type variable after class name

    public static <T> void clear(Stack<T> s) { // not shown }

    public static <T> void pushAll(Collection<T> src, Stack<T> dest) { }

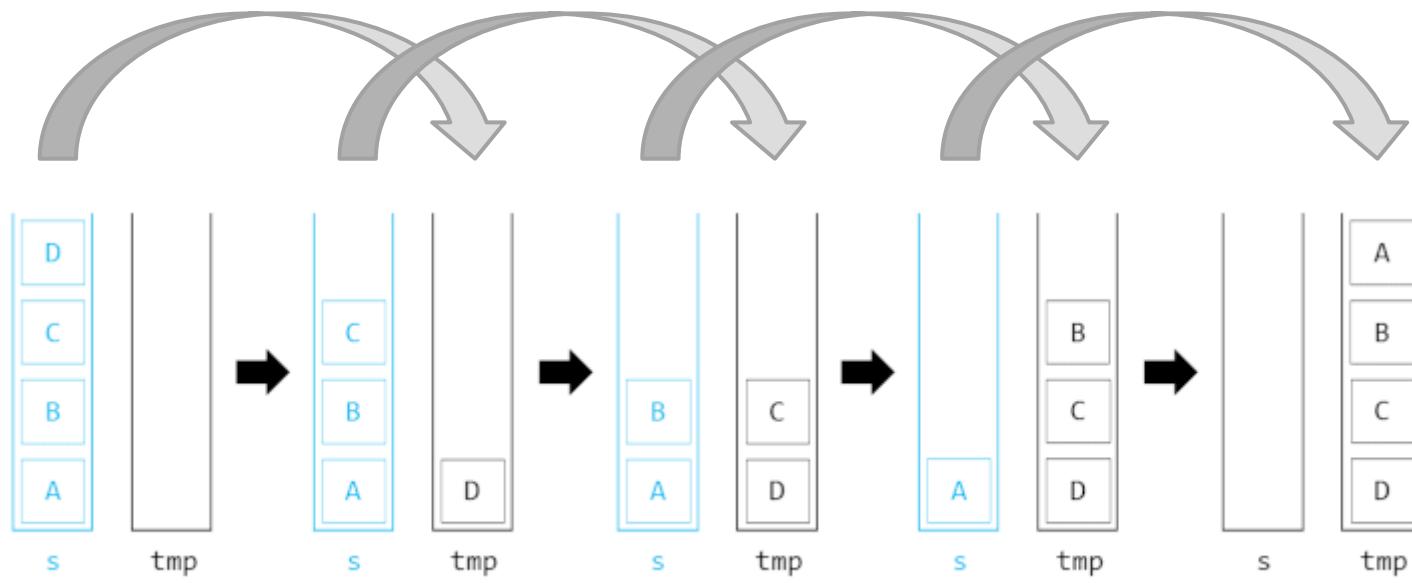
    // method is generic, type variable appears before return type
    public static <T> void popAll(Stack<T> src, Collection<T> dest) {
        while (src.size() > 0) {
            dest.add(src.pop());
        }
    }
}
```

Element types match

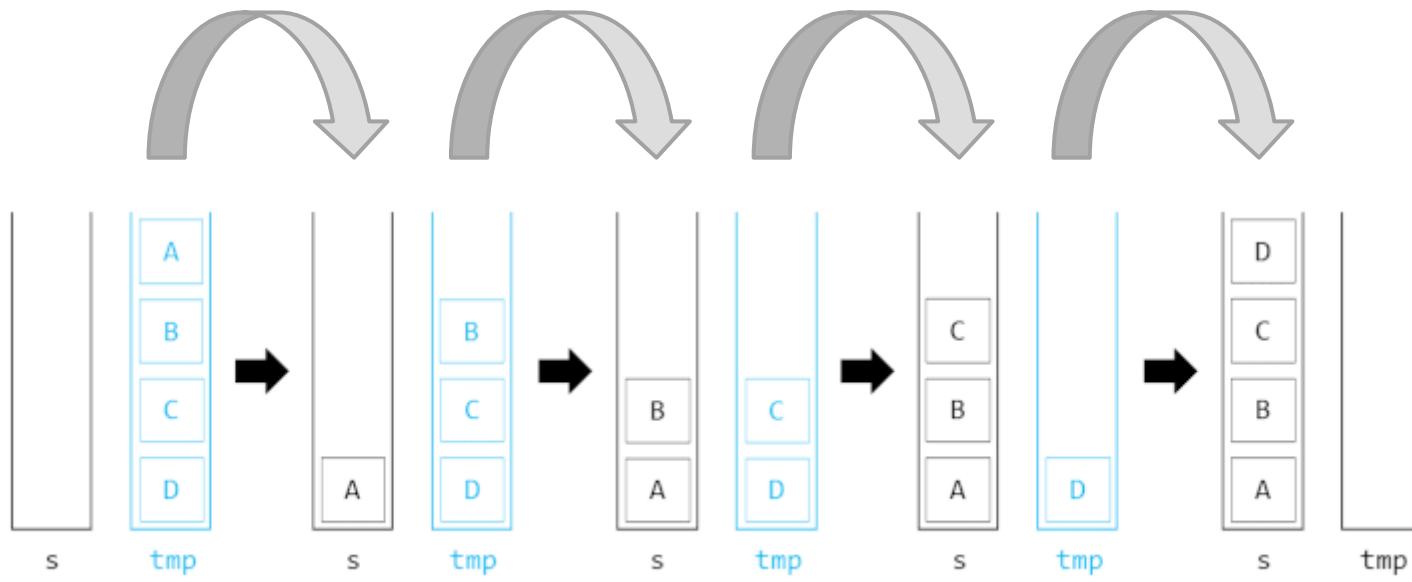
Defining a generic method

- ▶ the **contains** method tests if a stack contains a specified element
- ▶ inconvenient to implement without having access to the private fields of the stack because there is no public method(s) to iterate over the elements of a stack
- ▶ algorithm:
 - ▶ pop the elements of the input stack onto a temporary stack until the input stack is empty or we find the searched for element
 - ▶ pop each element of the temporary stack and push the element back onto the input stack

contains("Z")



contains("Z")



```
public static <T> boolean contains(Stack<T> s, Object obj) {  
    boolean result = false;  
    Stack<T> tmp = new ArrayStack<>();  
  
    // pop elements from s until obj is found (but don't pop an empty stack!)  
    // store popped elements in tmp  
    while (s.size() > 0 && !result) {  
        T elem = s.pop();  
        if (elem.equals(obj)) {  
            result = true;  
        }  
        tmp.push(elem);  
    }  
  
    // pop elements from tmp back onto s so that the state of  
    // s appears unchanged to the caller  
    while (tmp.size() > 0) {  
        s.push(tmp.pop());  
    }  
    return result;  
}  
}
```

Utility methods for queues

- ▶ we can implement similar generic methods that operate on queues
 - ▶ `clear(Queue<T> q)`
 - ▶ `enqueueAll(Collection<T> src, Queue<T> dest)`
 - ▶ `dequeueAll(Queue<T> src, Collection<T> dest)`
 - ▶ `contains(Queue<T>, Object obj)`

```
public class Queues { // class itself is not generic
                      // no type variable after class name

    // method is generic, type variable appears before return type
    public static <T> void clear(Queue<T> q) {
        while (q.size() > 0) {
            q.dequeue();
        }
    }
}
```

```
public class Queues { // class itself is not generic
    // no type variable after class name

    public static <T> void clear(Queue<T> q) { // not shown }

    // method is generic, type variable appears before return type
    public static <T> void enqueueAll(Collection<T> src, Queue<T> dest) {
        for (T elem : src) {
            dest.enqueue(elem);
        }
    }

}
```

Element types match

```
public class Queues { // class itself is not generic
    // no type variable after class name

    public static <T> void clear(Queue<T> q) { // not shown }

    public static <T> void enqueueAll(Collection<T> src, Queue<T> dest) { }

    // method is generic, type variable appears before return type
    public static <T> void dequeueAll(Stack<T> src, Collection<T> dest) {
        while (src.size() > 0) {
            dest.add(src.dequeue());
        }
    }
}
```

↑
↑
Element types match

```
public static <T> boolean contains(Queue<T> q, Object obj) {  
    boolean result = false;  
  
    // dequeue each element of q and then immediately enqueue  
    for (int i = 0; i < q.size(); i++) {  
        T elem = q.dequeue();  
        if (!result && elem.equals(obj)) {  
            result = true;  
        }  
        q.enqueue(elem);  
    }  
    return result;  
}  
  
}
```

Lists

List

- ▶ a list represents a finite collection of elements held in a linear sequence where each element can be accessed via its position (index) in the list
- ▶ we have already seen the Standard Library **List** interface and **ArrayList** class
- ▶ it is informative to study how lists can be implemented

A simplified list interface

- ▶ we will use a smaller interface than Java's **List** interface

Method	Summary
<code>size()</code>	returns the number of elements in the list
<code>isEmpty()</code>	returns true if the list has no elements
<code>get(int index)</code>	returns the element at the specified index
<code>set(int index, E elem)</code>	replaces the element at the specified index
<code>add(E elem)</code>	adds an element to the end of the list
<code>add(int index, E elem)</code>	inserts an element at the specified index
<code>remove(int index)</code>	removes and returns the element at the specified index
<code>iterator()</code>	returns an iterator over the elements of the list

```
import java.util.Iterator;

public interface SList<E> extends Iterable<E>  {

    public int size();

    default public boolean isEmpty() {
        return this.size() == 0;
    }

    public void add(E elem);

    public E get(int index);

    public E set(int index, E elem);

    public void add(int index, E elem);

    public E remove(int index);

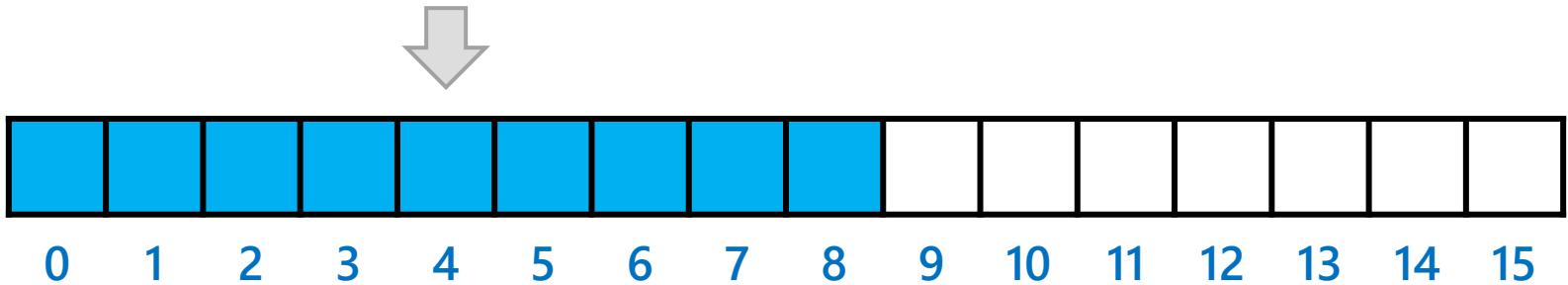
    public Iterator<E> iterator();
}
```

An array-based generic list

- ▶ very similar to an array-based generic stack
- ▶ inserting an element into the list and removing an element from the list requires some explanation

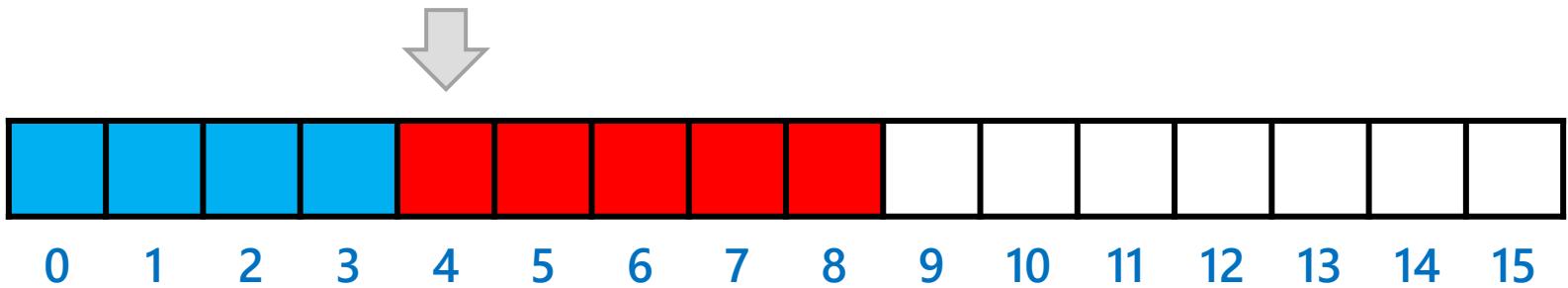
Insertion using an index

- ▶ inserting an element into a list at a specified index requires moving the current elements starting at the specified index one position to the right
- ▶ e.g., insert at index 4



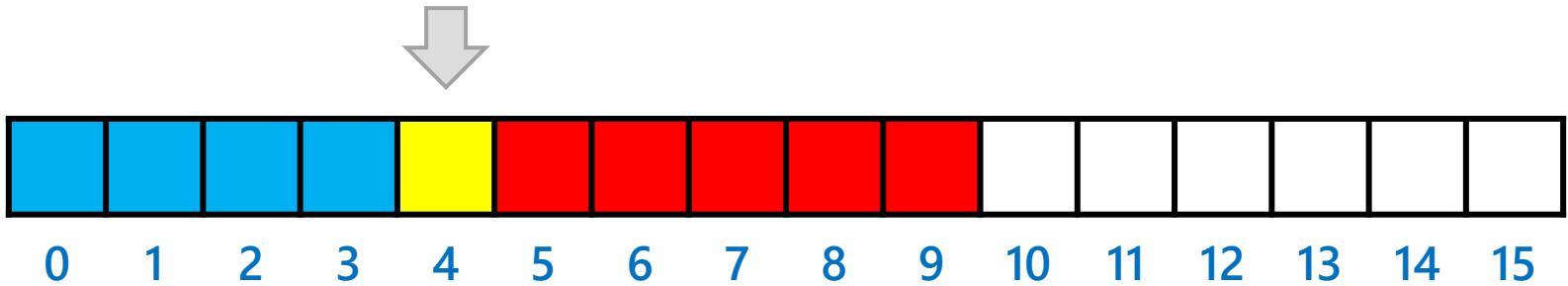
Insertion using an index

- ▶ shifting the elements in red one position to the right...



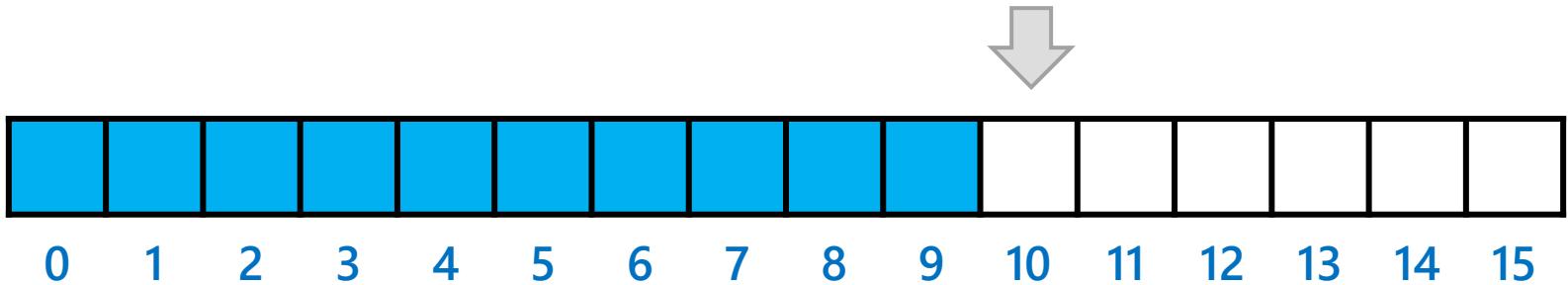
Insertion using an index

- ... makes room to insert the new element (shown in yellow)



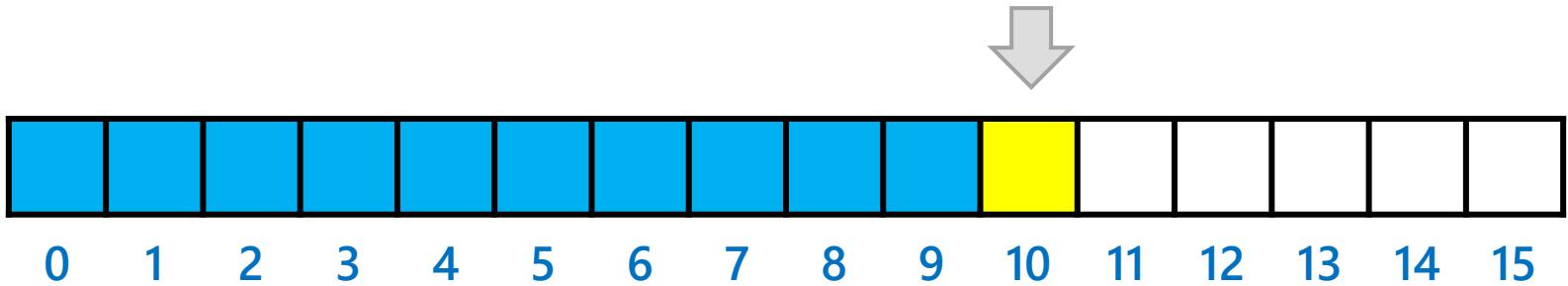
Insertion using an index

- inserting at the end of the list is usually allowed
- slightly unusual because the index is not technically valid



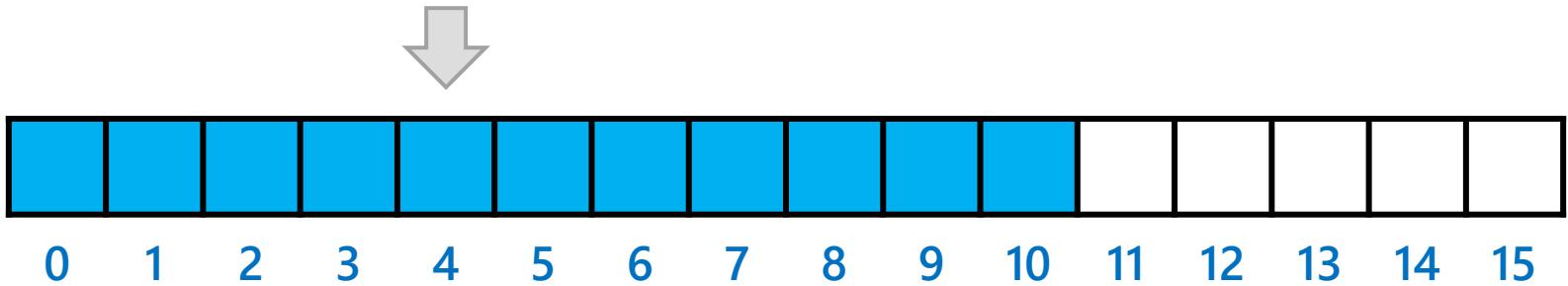
Insertion using an index

- ▶ can be implemented simply by calling the **add(E elem)** method



Removing using an index

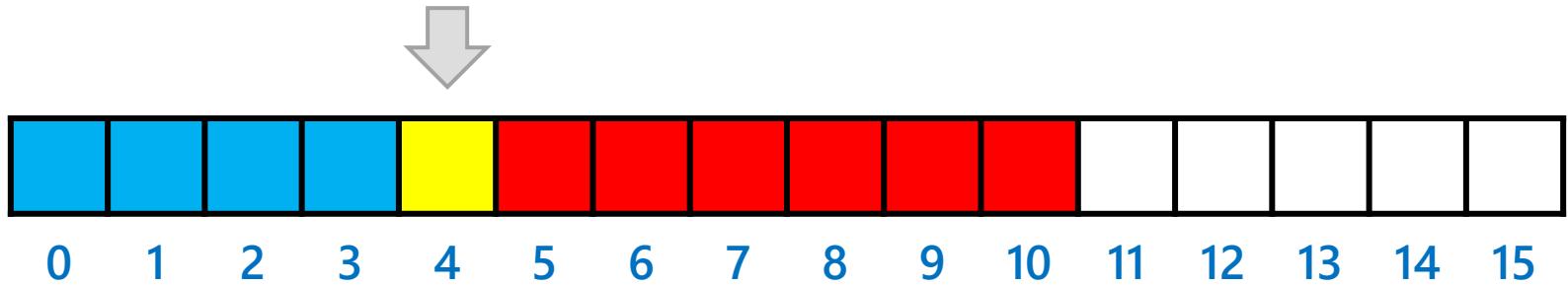
- removing an element at a specified index i requires moving the current elements starting at $(i + 1)$ one position to the left



- the removed element is usually returned to the caller

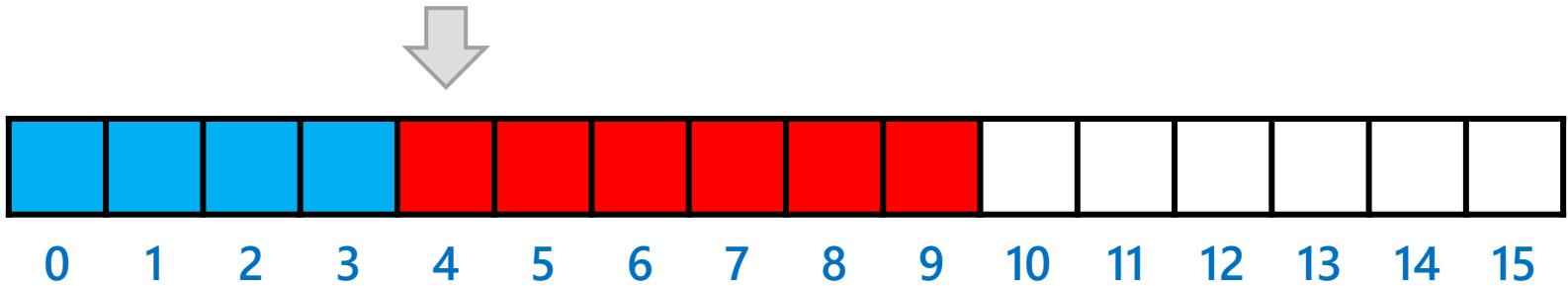
Removing using an index

- ▶ first copy the element shown in yellow so that it can be returned to the caller, then
- ▶ shifting the elements in red one position to the left...



Removing using an index

- ... removes the element at the specified index



Array-based generic list (cont)

Inserting and removing

- ▶ the previous illustrations should make it clear that inserting or removing an element in an array-based list has worst-case complexity in $O(n)$ because we have to move up to $n - 1$ elements

```
public class SArrayList<E> implements SList<E> {  
  
    private final int DEFAULT_CAPACITY = 16;  
    private Object[] arr;  
    private int size;  
  
    public SArrayList() {  
        this.arr = new Object[DEFAULT_CAPACITY];  
        this.size = 0;  
    }  
  
    @Override  
    public int size() {  
        return this.size;  
    }  
}
```

```
@Override  
public void add(E elem) {  
    // do we need to resize the array?  
    if (this.size() == this.arr.length) {  
        this.arr = Arrays.copyOf(this.arr,  
                               this.arr.length * 2);  
    }  
    this.arr[this.size] = elem;  
    this.size++;  
}
```

```
/**  
 * Throws an {@code IndexOutOfBoundsException} if index is  
 * less than 0 or greater than {@code this.size - 1}.  
 *  
 * @param index an index to validate  
 * @throws {@code IndexOutOfBoundsException} if index  
 *         is less than 0 or greater than {@code this.size - 1}  
 */  
private void checkIndex(int index) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException(  
            "negative index: " + index);  
    }  
    else if (index >= this.size) {  
        throw new IndexOutOfBoundsException(  
            "index out of bounds: " + index + ", size: " +  
            this.size);  
    }  
}
```

```
@Override  
public E get(int index) {  
    this.checkIndex(index);  
    return (E) this.arr[index];  
}  
  
@Override  
public E set(int index, E elem) {  
    // get element at index, this also checks the index  
    E old = this.get(index);  
    this.arr[index] = elem;  
    return old;  
}
```

```
@Override
public void add(int index, E elem) {
    // index can be equal to this.size
    if (index == this.size) {
        this.add(elem);
        return;
    }
    this.checkIndex(index);

    // move elements at indexes size-1, size-2, ..., index
    // one position to right
    // i is in the index of the element to move to the right
    for (int i = this.size - 1; i >= index; i--) {
        this.arr[i + 1] = this.arr[i]; // can this throw?
    }
    // insert elem and update size
    this.arr[index] = elem;
    this.size++;
}
```

There is an error in this method; can you see what it is?

```
@Override
public E remove(int index) {
    // get element at index, this also checks the index
    E removed = this.get(index);

    // move elements at indexes index+1, index+2, ...
    // size-1 one position to left
    // i is in the index of the element to move to the right
    for (int i = index + 1; i < this.size; i++) {
        this.arr[i - 1] = this.arr[i]; // can this throw?
    }
    // null out old last element and update size
    this.arr[this.size - 1] = null;
    this.size--;
    return removed;
}
```

```
@Override
public Iterator<E> iterator() {
    return new ArrayIterator();
}

private class ArrayIterator implements Iterator<E> {

}

}
```

Iterators for array-based lists

- ▶ review **Iterator** interface from *Lecture 20: Interfaces*

The Iterator interface

- ▶ an **Iterator** is an object that knows how to iterate over a collection of elements or a sequence of values
- ▶ using an **Iterator** object is the only guaranteed safe way of iterating over a **List** or **Set** and removing elements during the iteration
 - ▶ see **List** and **Set** notebooks for details
- ▶ Java's for-each loop actually uses an **Iterator** object
 - ▶ the language hides the object from the programmer by having the compiler create the object and call the appropriate methods

The Iterator interface

```
package java.util;

public interface Iterator<T> {

    public boolean hasNext();
    public T next();
    public default void remove() {
        throw new UnsupportedOperationException();
    }

    // one more default method not shown here
}
```

The **remove** method

- ▶ the **remove** method removes the element that was most recently returned by **next**
 - ▶ it can be called only once for each call to **next**
 - ▶ an **IllegalStateException** is thrown if **remove** is called without first calling **next**, or if **remove** has already been called after the most recent call to **next**
- ▶ using an iterator is the only safe way to filter a collection
 - ▶ e.g., to remove all negative values from a list or set of integers

```
// assume t refers to a List or Set
for (Iterator<Integer> iter = t.iterator(); iter.hasNext(); ) {
    Integer val = iter.next();
    if (val < 0) {
        iter.remove();      // can call remove once for each call to next
    }
}
```

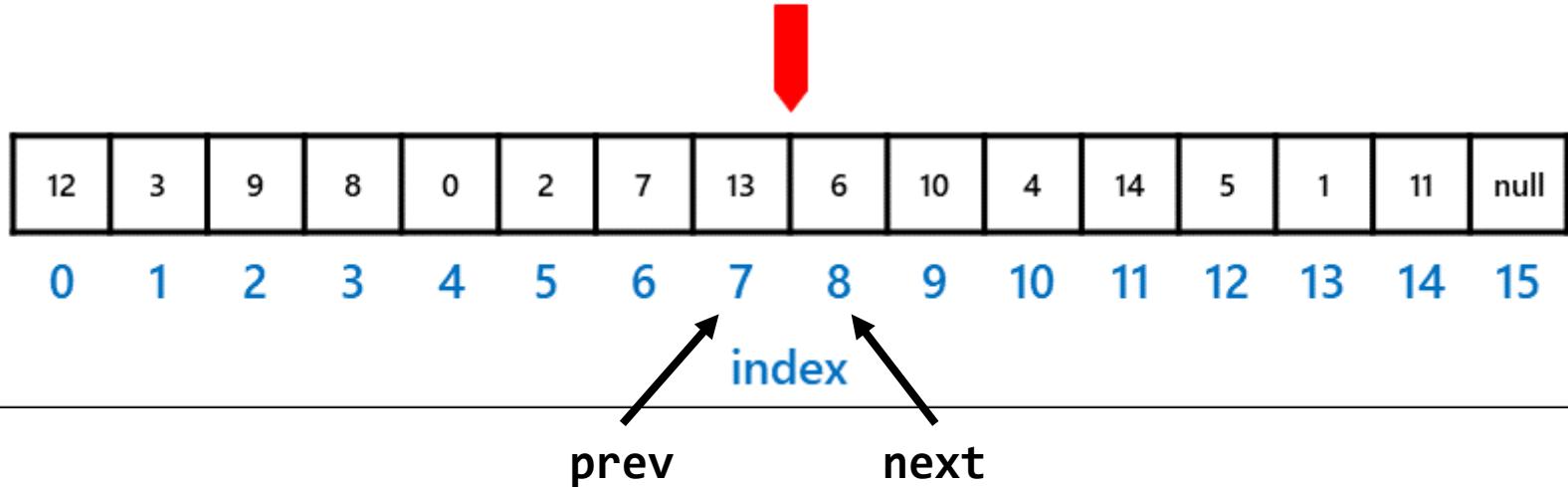
An iterator is similar to a cursor

- ▶ in a text editor, a cursor sits between letters

```
private class ArrayIterator implements Iterator<E>
```

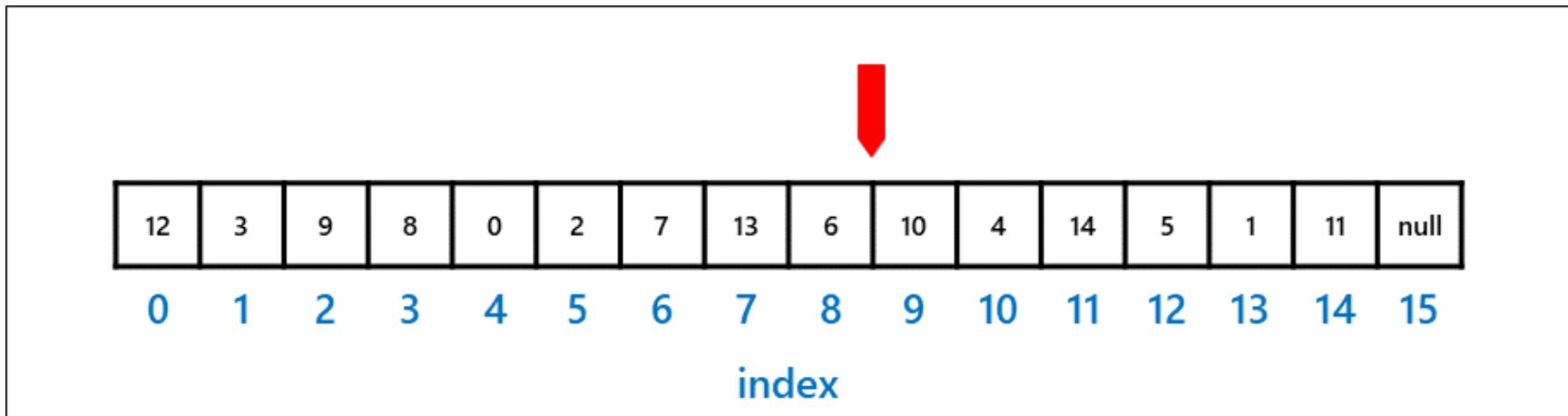
Example of an iterator

- iterator between indexes 7 and 8



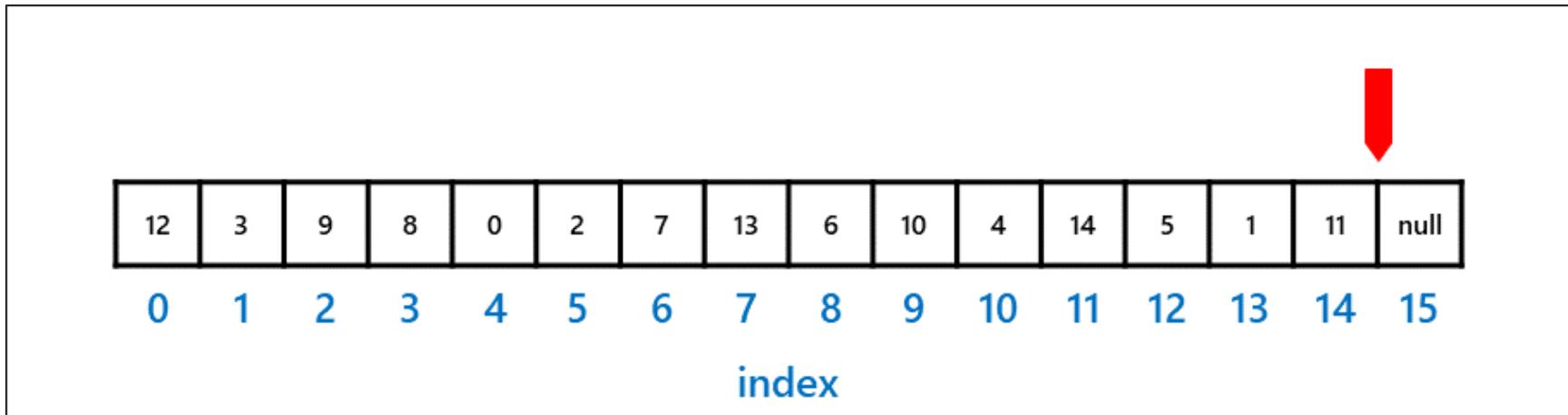
- calling **next** returns the element **6** and advances the iterator

Iterator after calling next



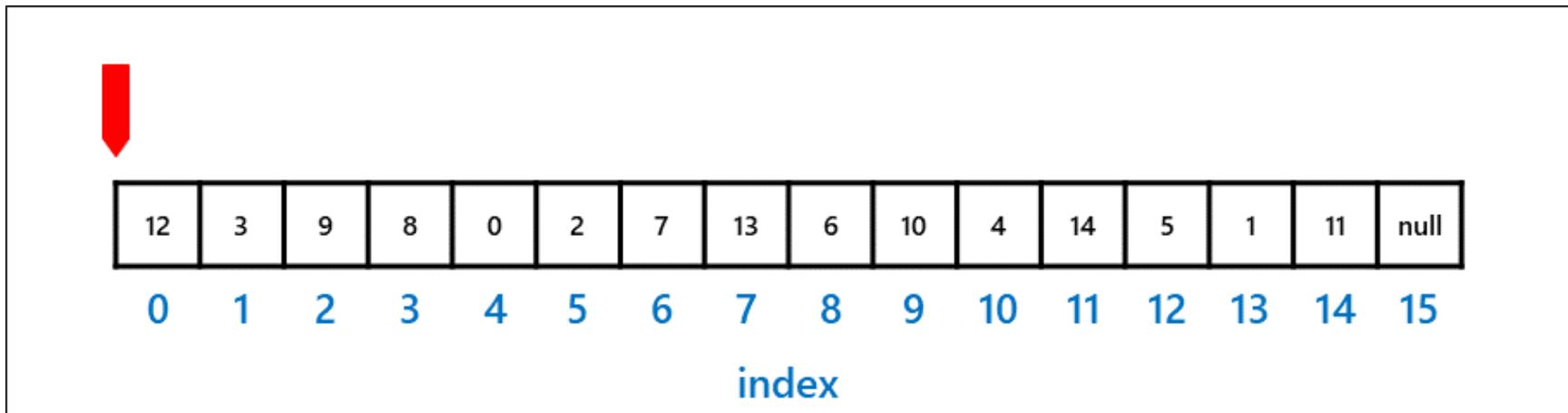
Iterator at end of list

- ▶ **hasNext** return false
- ▶ **next** will throw an exception



Iterator at start of list

- at the start of the list, there is no previous element so we set **prev = -1**



```
private class ArrayIterator implements Iterator<E> {  
    /**  
     * Index of element returned by subsequent call to next.  
     */  
    private int next;  
  
    /**  
     * Index of element returned by most recent call  
     * to next. Reset to -1 if this element is deleted by a  
     * call to remove.  
     */  
    private int prev;  
  
    ArrayIterator() {  
        this.next = 0;  
        this.prev = -1;  
    }  
}
```

A nested, inner class (or just inner class).

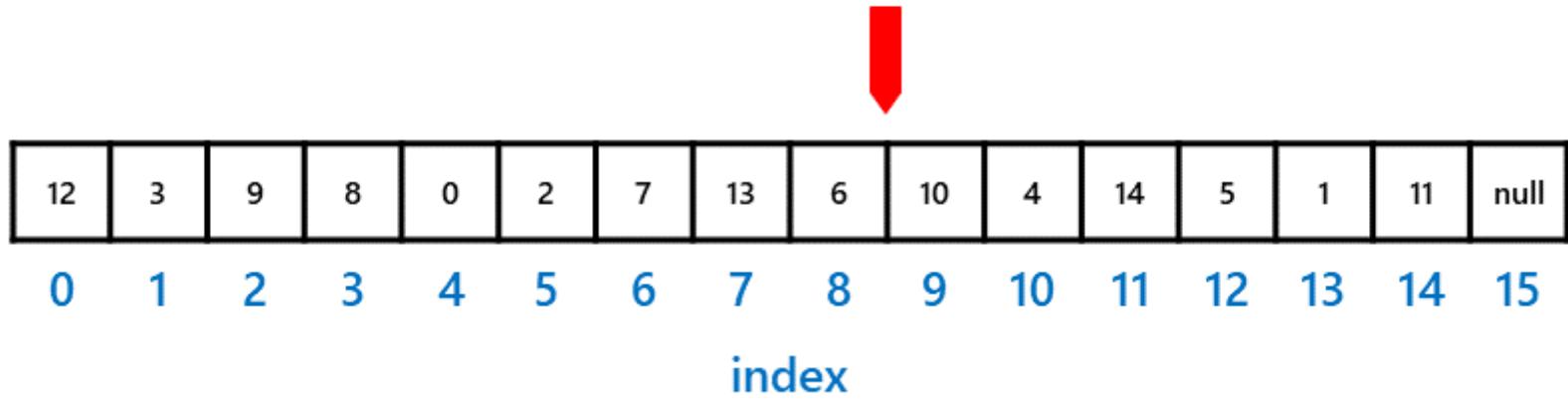
An inner class object has access to the members of its enclosing object, so **ArrayIterator** has access to the fields and methods of **SArrayList**.

```
@Override  
public boolean hasNext() {  
    return this.next < SArrayList.this.size;  
}
```

```
@Override
public E next() {
    if (!this.hasNext()) {
        throw new NoSuchElementException();
    }
    E e = SArrayList.this.get(this.next);
    // slightly more efficient would be
    // E e = SArrayList.this.arr[this.next];
    this.prev = this.next;
    this.next++;
    return e;
}
```

Iterator remove

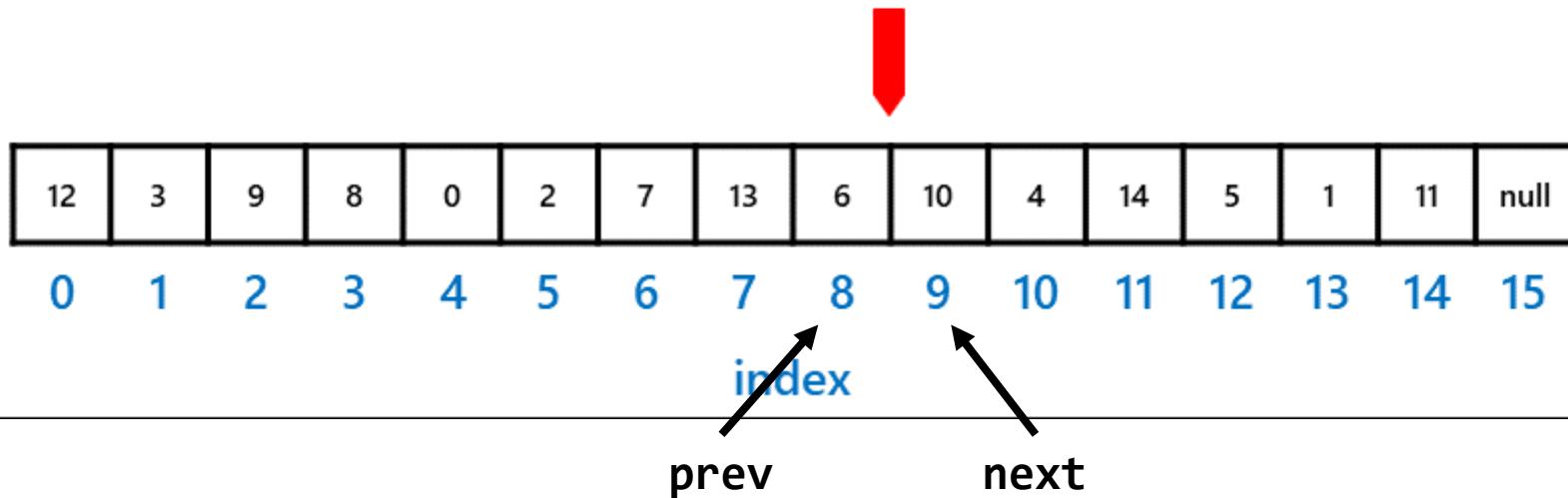
- ▶ suppose that we have the following iterator



- ▶ the most recent element returned by **next** is the **6**
- ▶ the element that should be returned by the next call to **next** is the **10**

Iterator remove

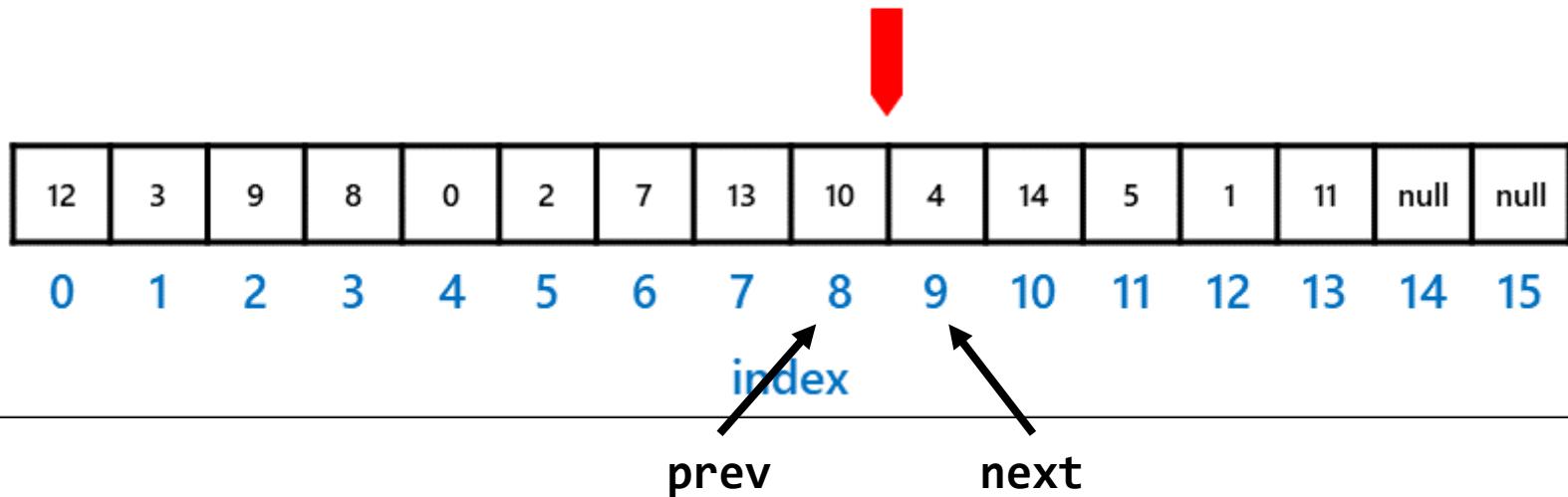
- ▶ calling **remove** removes the 6



- ▶ that is, we remove the element at index **prev** by calling the **SArrayList remove** method

Iterator remove

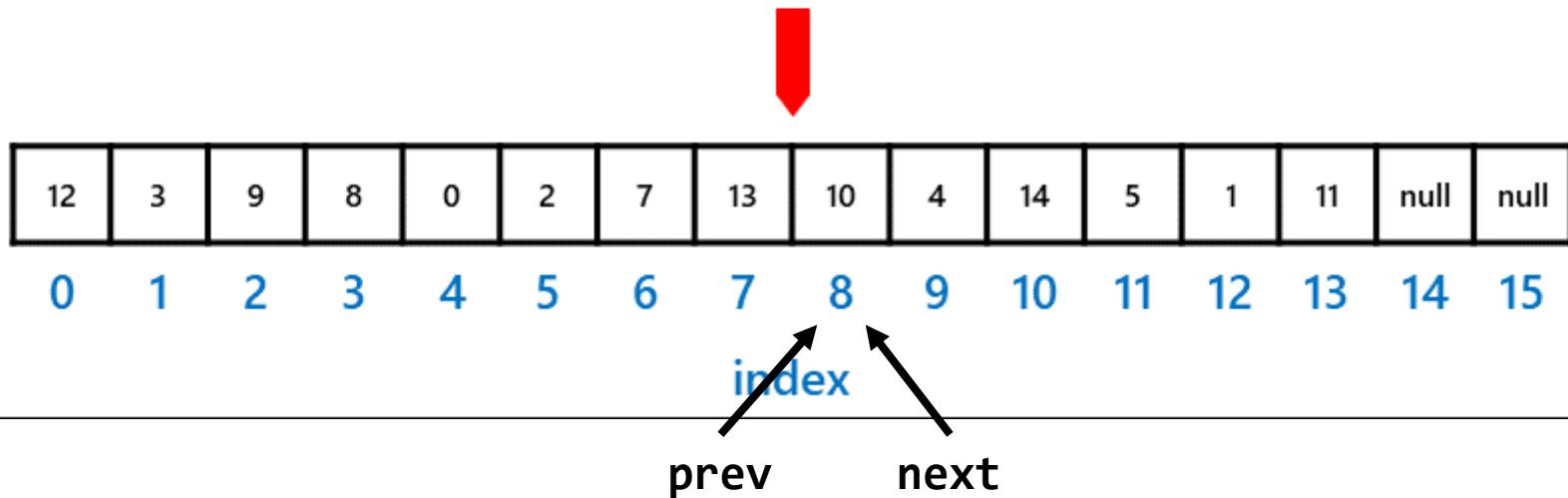
- ▶ after removing the **6** we have the following situation



- ▶ notice that **next** is now the wrong index (it should be the index of the element **10**)
- ▶ easy to fix, just decrement **next**

Iterator remove

- ▶ **next** and **prev** now have the same value which does not make sense



- ▶ we set **prev = -1** to indicate that we cannot call **remove** again without first calling **next**

```
@Override
public void remove() {
    if (this.prev == -1) {
        throw new IllegalStateException();
    }
    SArrayList.this.remove(this.prev);
    this.next--;
    this.prev = -1;
}

} // end ArrayIterator

} // end SArrayList
```

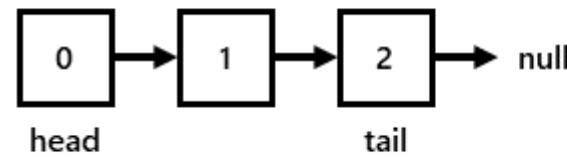
(Singly) Linked lists

Linked lists

- ▶ a linked list is an implementation of a list that uses linked nodes to represent elements in a sequence
- ▶ the node at the front of the list is traditionally called the *head* node and the node at the end of the list is called the *tail* node
- ▶ in a singly-linked list, each node has a link to the next node in the sequence
- ▶ in a doubly-linked list, each node has links to the next and previous nodes in the sequence

Visualizing a linked list

- looks similar to a node-based stack or queue



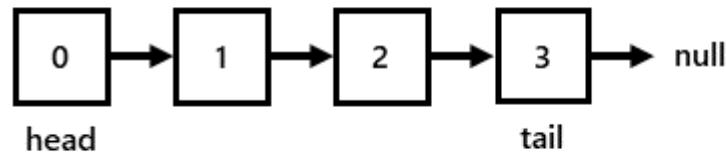
The empty list

head = null

tail = null

Linked lists have a recursive structure

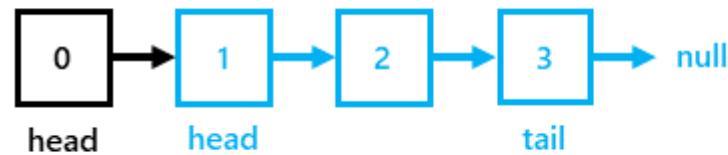
- every node of a linked list can be thought of as being the head node of some list



- this makes implementing **subList** very easy

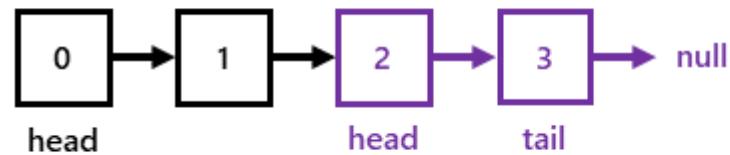
Linked lists have a recursive structure

- the second node is the head node of a list of size 3



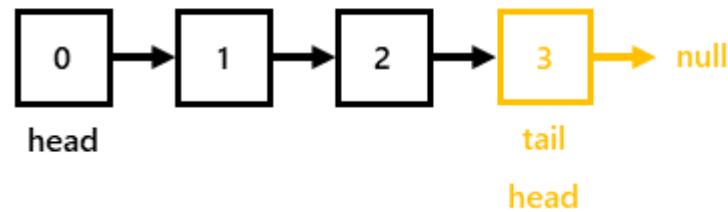
Linked lists have a recursive structure

- the third node is the head node of a list of size 2



Linked lists have a recursive structure

- ▶ the tail node is the head node of a list of size 1



Linked lists have a recursive structure

- ▶ because the structure of a linked list is recursive, most linked-list algorithms can be implemented recursively
- ▶ for example, we can easily compute the size of linked list recursively

```
public int size() {  
    return this.recSize(this.head);  
}  
  
private int recSize(Node<E> head) {  
    if (head == null) {  
        return 0;  
    }  
    // at least 1 node in this list (the head node)  
    // recursively compute size of the rest of the list  
    return 1 + this.recSize(head.next);  
}
```

Exam

- ▶ 2021/12/21, Tuesday
2:00PM - 5:00PM
Gym 3 (Bartlett) Mitchell Hall

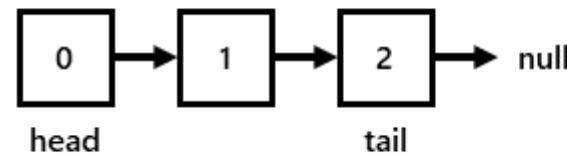
(Singly) Linked lists

Linked lists

- ▶ a linked list is an implementation of a list that uses linked nodes to represent elements in a sequence
- ▶ the node at the front of the list is traditionally called the *head* node and the node at the end of the list is called the *tail* node
- ▶ in a singly-linked list, each node has a link to the next node in the sequence
- ▶ in a doubly-linked list, each node has links to the next and previous nodes in the sequence

Visualizing a linked list

- looks similar to a node-based stack or queue



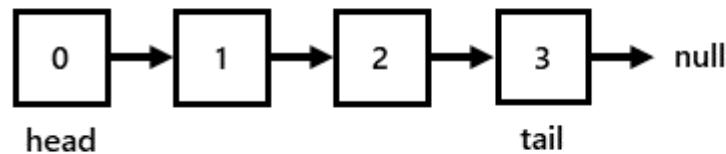
The empty list

head = null

tail = null

Linked lists have a recursive structure

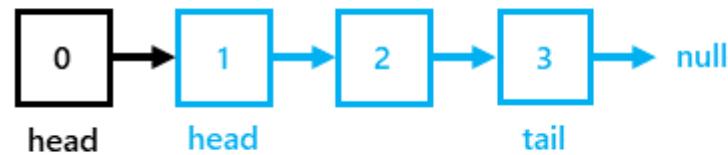
- every node of a linked list can be thought of as being the head node of some list



- this makes implementing **subList** very easy

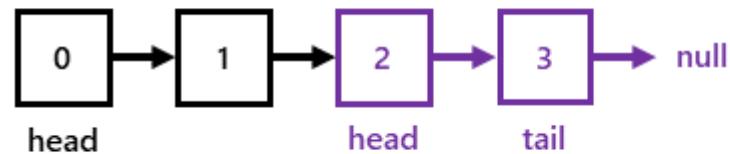
Linked lists have a recursive structure

- the second node is the head node of a list of size 3



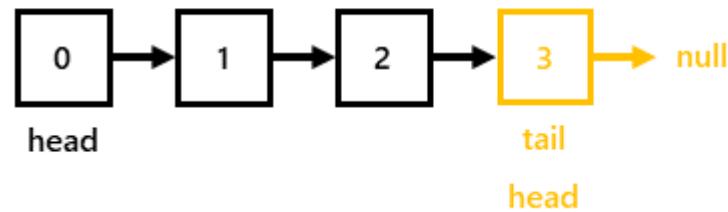
Linked lists have a recursive structure

- the third node is the head node of a list of size 2



Linked lists have a recursive structure

- the tail node is the head node of a list of size 1



Linked lists have a recursive structure

- ▶ because the structure of a linked list is recursive, most linked-list algorithms can be implemented recursively
- ▶ for example, we can easily compute the size of linked list recursively

```
public int size() {  
    return this.recSize(this.head);  
}  
  
private int recSize(Node<E> head) {  
    if (head == null) {  
        return 0;  
    }  
    // at least 1 node in this list (the head node)  
    // recursively compute size of the rest of the list  
    return 1 + this.recSize(head.next);  
}
```

SLinkedList<E>

- ▶ a generic, singly-linked list class
- ▶ has the same generic **Node** class used in **LinkedStack** and **LinkedQueue**
 - ▶ **Node** class is package private so that it is possible to create a utility class having methods that can use nodes
- ▶ has package private methods that return the head and tail nodes, and a node by index of the list
 - ▶ so that the utility class methods can access the nodes of the linked list

```
public class SLinkedList<E> implements SList<E> {

    static class Node<E> {}

    /**
     * The number of elements in the linked list.
     */
    private int size;

    /**
     * The first node of the linked list; will be <code>null</code> for an empty
     * list.
     */
    private Node<E> head;

    /**
     * The last node of the linked list; will be <code>null</code> for an empty
     * list.
     */
    private Node<E> tail;
```

```
/**  
 * Returns the head node of this list.  
 *  
 * @return the head node of this list  
 */  
Node<E> head() {  
    return this.head;  
}  
  
/**  
 * Returns the tail node of this list.  
 *  
 * @return the tail node of this list  
 */  
Node<E> tail() {  
    return this.tail;  
}
```

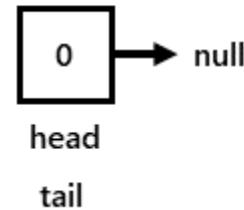
```
/**  
 * Initialize an empty list.  
 */  
public SLinkedList() {  
    this.size = 0;  
    this.head = null;  
    this.tail = null;  
}  
  
/**  
 * Get the number of elements in the list.  
 *  
 * @return the number of elements in the list.  
 */  
@Override  
public int size() {  
    return this.size;  
}
```

Adding to the end of the list

- ▶ identical to **enqueue** operation for a queue
 - ▶ the **enqueue** implementation in the lecture slides and notebook has an unnecessary special case

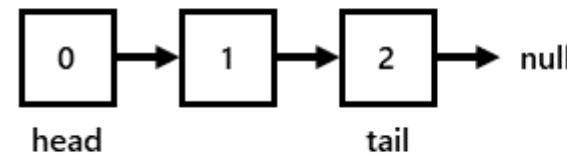
Adding to the end of the list

- ▶ adding to the end of an empty list is a special case
 - ▶ both **head** and **tail** must be modified



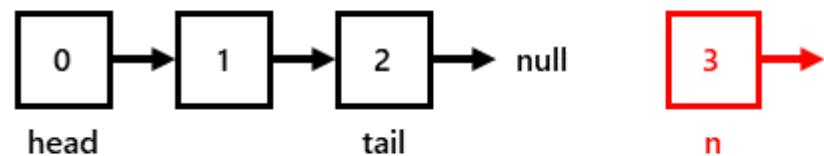
Adding to the end of the list

- ▶ adding to the end of a non-empty list involves modifying only **tail**



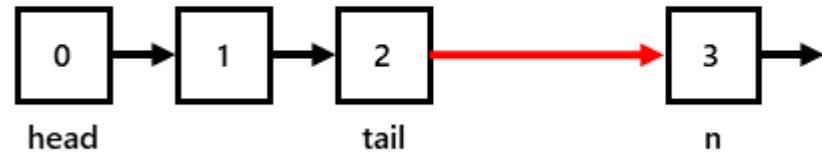
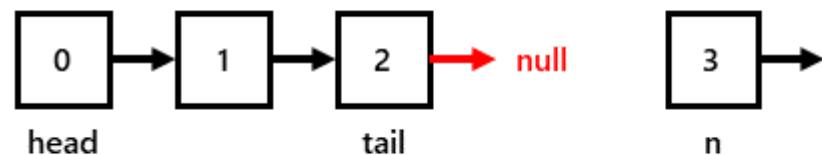
Adding to the end of the list

- ▶ first, create a node to hold the element to add



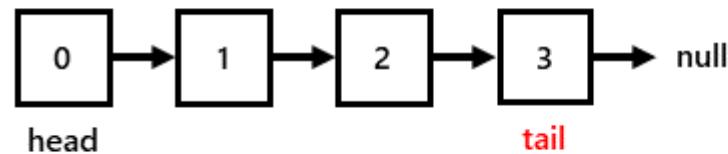
Adding to the end of the list

- ▶ next, link the current tail node to the new node



Adding to the end of the list

- ▶ finally, update **tail** so that it refers to the newly added node



```
/**  
 * Adds the given element to the end of the list.  
 *  
 * @param elem the element to add  
 */  
  
@Override  
public void add(E elem) {  
    Node<E> n = new Node<>(elem, null);  
    if (this.size == 0) {  
        this.head = n;  
    }  
    else {  
        this.tail.next = n;  
    }  
    this.tail = n;  
    this.size++;  
}
```

Getting and setting by index

- ▶ performing an operation that uses an index requires that we move to the node at (or near) the index
 - ▶ because we have only a reference to the head node
- ▶ it is useful to write methods that can:
 - ▶ validate an index
 - ▶ move to a node at a specified index

```
/**  
 * Validates the specified index.  
 *  
 * @param index an index  
 * @throws IndexOutOfBoundsException if  
 *      {@code index < 0 || index >= this.size()}  
 */  
  
void validate(int index) {  
    if (index < 0 || index >= this.size) {  
        throw new IndexOutOfBoundsException(  
            "index out of bounds: " + index);  
    }  
}
```

```
/**  
 * Returns the node at the specified index. Assumes that the index is  
 * valid for this list to avoid re-validating the index.  
 *  
 * @param index a valid index for this list  
 * @return the node at the specified index  
 */  
  
Node<E> moveTo(int index) {  
    if (index == this.size - 1) {  
        return this.tail;  
    }  
    Node<E> n = this.head;  
    // follow index next links  
    for (int i = 0; i < index; i++) {  
        n = n.next;  
    }  
    return n;  
}
```

```
/**  
 * Alternative implementation using recursion.  
 */  
  
Node<E> moveTo(int index) {  
    if (index == this.size - 1) {  
        return this.tail;  
    }  
    return moveTo(this.head, index);  
}  
  
  
private static <E> Node<E> moveTo(Node<E> n, int index) {  
    if (index == 0) {  
        return n;  
    }  
    return moveTo(n.next, index - 1);  
}
```

get and set

- ▶ given the **moveTo** method, **get** and **set** are trivial to implement
- ▶ notice that the complexity of **moveTo** is in $O(n)$ because it follows up to $(n - 2)$ links in the worst case

```
/**  
 * Returns the element at the specified position in the list.  
 *  
 * @param index index of the element to return  
 * @return the element at the specified position  
 * @throws IndexOutOfBoundsException if the index is out of the range  
 *                                     {@code (index < 0 || index >= size())}  
 */  
  
@Override  
public E get(int index) {  
    this.validate(index);  
    Node<E> n = this.moveTo(index);  
    return n.elem;  
}
```

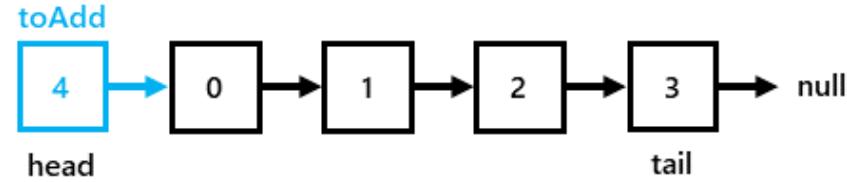
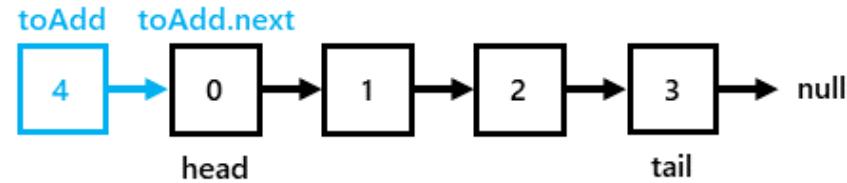
```
/**  
 * Sets the element at the specified position in the list.  
 *  
 * @param index index of the element to set  
 * @param elem element to be stored at the specified position  
 * @throws IndexOutOfBoundsException if the index is out of the range  
 *                                     {@code (index < 0 || index >= size())}  
 */  
  
@Override  
public E set(int index, E elem) {  
    this.validate(index);  
    Node<E> n = this.moveTo(index);  
    E old = n.elem;  
    n.elem = elem;  
    return old;  
}
```

Inserting at the front of the list

- ▶ inserting an element at the front of the list is identical to pushing an element onto the top of a node-based stack
- ▶ has complexity in $O(1)$ compared to $O(n)$ for an array-based list

Inserting at the front of the list

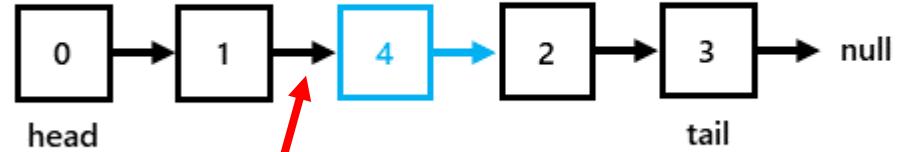
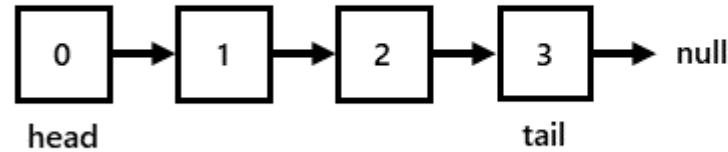
- ▶ create a new node **toAdd** and link it to the head of the list
- ▶ then update head to refer to the new node



```
/**  
 * Adds an element to the front of this list.  
 *  
 * @param elem the element to add  
 */  
  
public void addFront(E elem) {  
    Node<E> toAdd = new Node<>(elem, null);  
    toAdd.next = this.head;  
    this.head = toAdd;  
    this.size++;  
}
```

Inserting into the list using an index

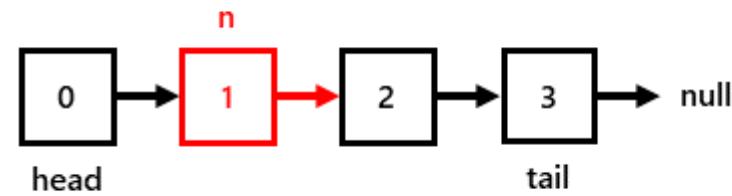
- inserting an element at index i requires adjusting the next link of the node at index $(i - 1)$
- e.g., insert the element **4** at index **2** of the following list



link that needs adjusting

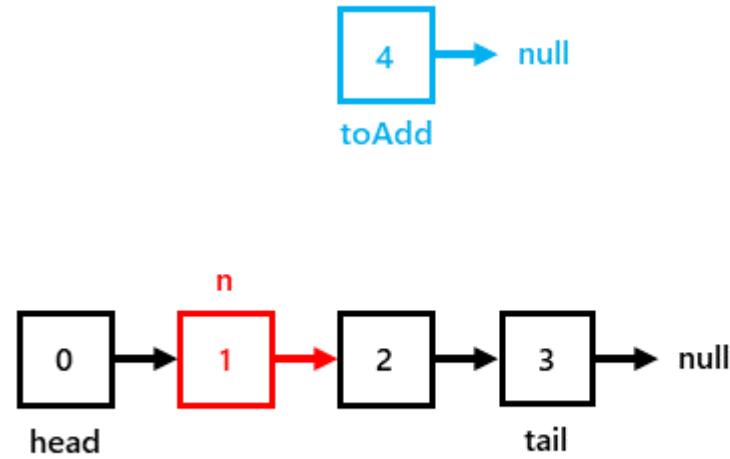
Inserting into the list using an index

- ▶ first move to the node located at **index - 1**



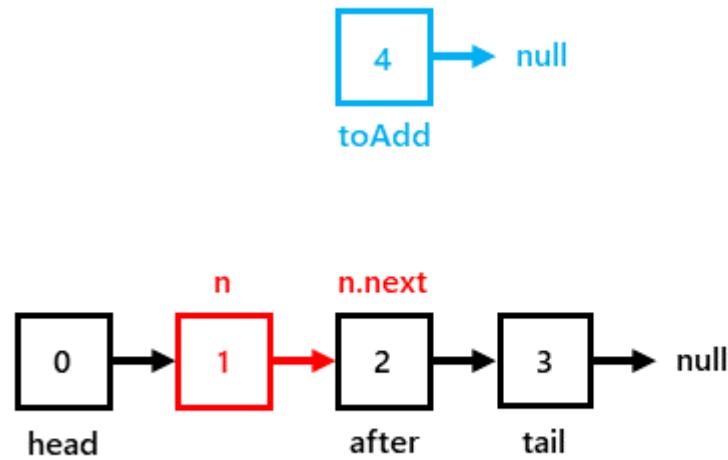
Inserting into the list using an index

- create a new node for the element that is being inserted



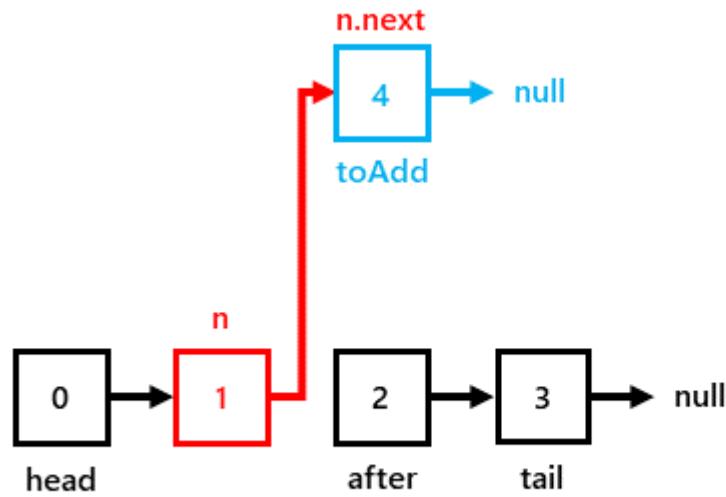
Inserting into the list using an index

- ▶ get a reference to the node currently at position **index**
 - ▶ this node will become the node after the inserted node



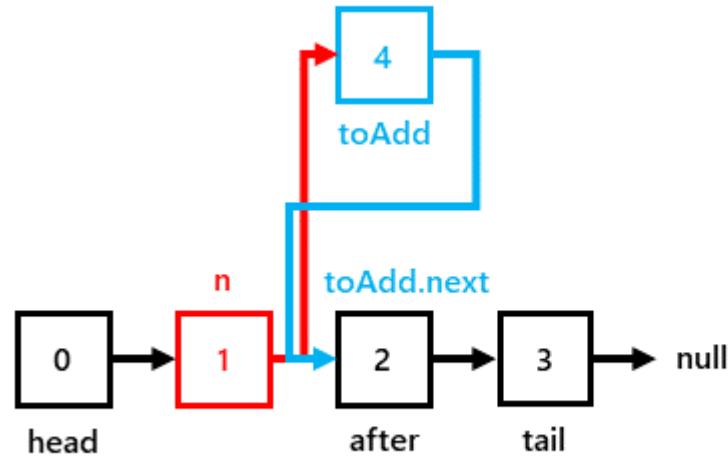
Inserting into the list using an index

- link the node at position **index - 1** to the new node



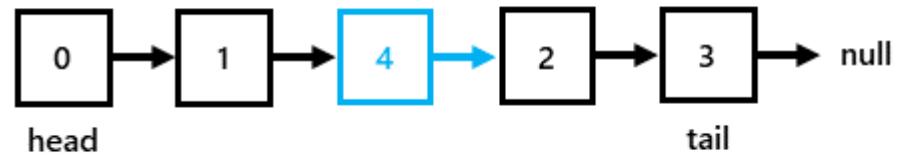
Inserting into the list using an index

- link the new node to the remainder of the list



Inserting into the list using an index

- increment the size of the list and return



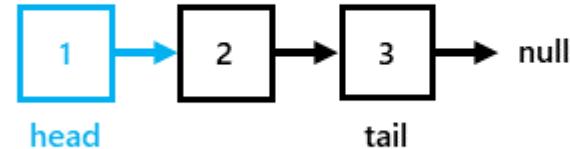
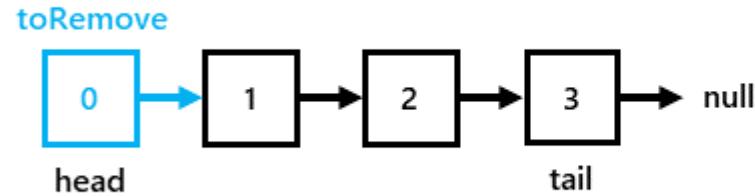
```
/**  
 * Inserts an element at the specified index of this list. Shifts the element  
 * currently at that position (if any) and any subsequent elements to the right.  
 *  
 * @param index the index at which to add the element  
 * @param elem the element to add  
 * @throws IndexOutOfBoundsException if the index is out of the range  
 *                                     {@code (index < 0 || index > size())}  
 */  
  
@Override  
public void add(int index, E elem) {  
    if (index == this.size) {  
        this.add(elem);  
        return;  
    }  
    this.validate(index);  
    if (index == 0) {  
        this.addFront(elem);  
    } else {  
        Node<E> n = this.moveTo(index - 1);  
        Node<E> toAdd = new Node<>(elem, n.next);  
        n.next = toAdd; this.size++;  
    }  
}
```

Removing the front element

- ▶ removing an element at the front of the list is identical to dequeuing an element from a node-based queue
- ▶ has complexity in $O(1)$ compared to $O(n)$ for an array-based list

Removing the front element

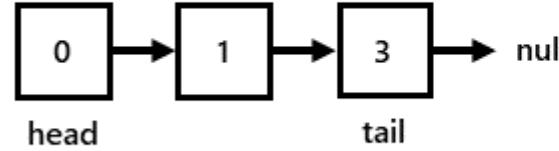
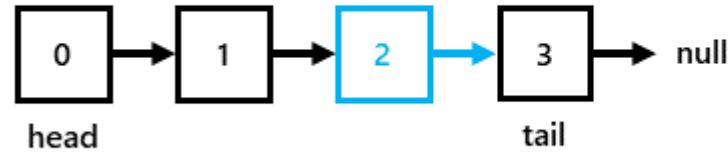
- ▶ first obtain a reference to the node to be removed
- ▶ then update the head node, decrement the size, and return the element in the removed node



```
/**  
 * Removes the first element of this list and returns the element.  
 *  
 * @return the removed element  
 * @throws NoSuchElementException if the list is empty  
 */  
  
public E removeFront() {  
    if (this.size == 0) {  
        throw new NoSuchElementException("list is empty");  
    }  
    Node<E> toRemove = this.head;  
    this.head = toRemove.next;  
    // special case of removing from a list of size 1  
    if (this.size == 1) {  
        this.tail = null;  
    }  
    this.size--;  
    return toRemove.elem;  
}
```

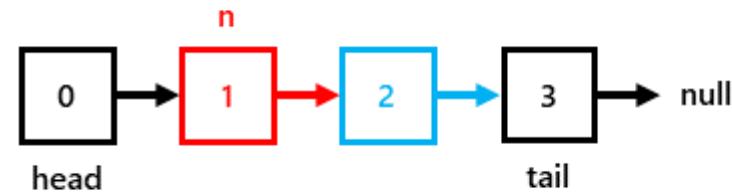
Removing an element by index

- removing an element by index is similar to inserting an element at an index
- move to the node at **index - 1** and then remove the following node



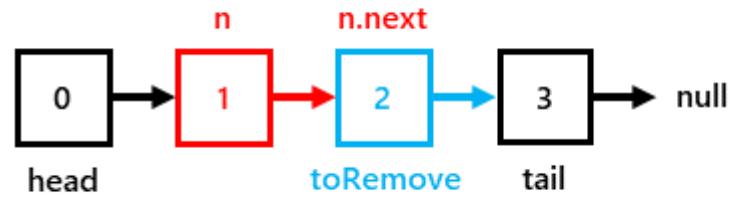
Removing an element by index

- ▶ first, move to the node at **index - 1**



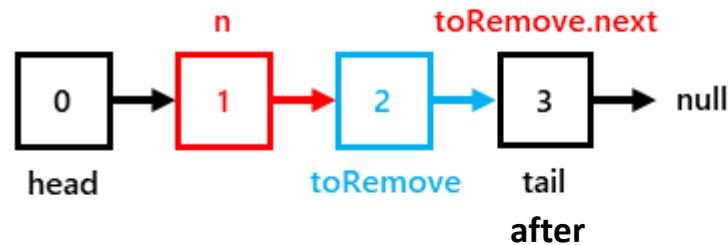
Removing an element by index

- ▶ create a variable **toRemove** that refers to the node to be removed



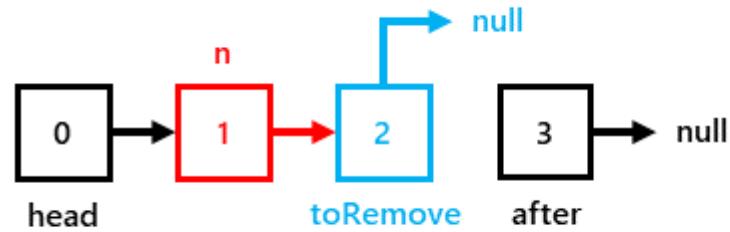
Removing an element by index

- ▶ create a variable **after** that refers to the node after the removed element (head node of the rest of the list)



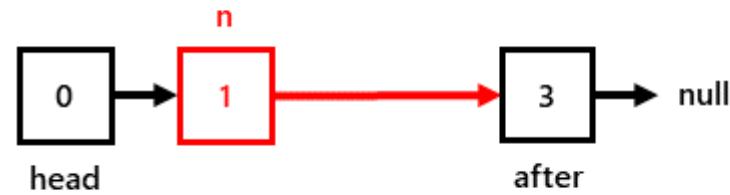
Removing an element by index

- ▶ unlink **toRemove** from the rest of the list



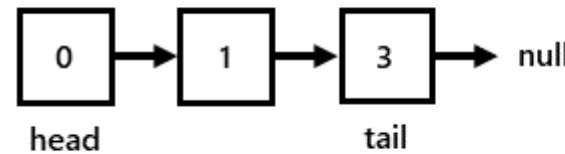
Removing an element by index

- ▶ link node **n** to **after**



Removing an element by index

- ▶ finally, decrement the size of the list and return the removed element

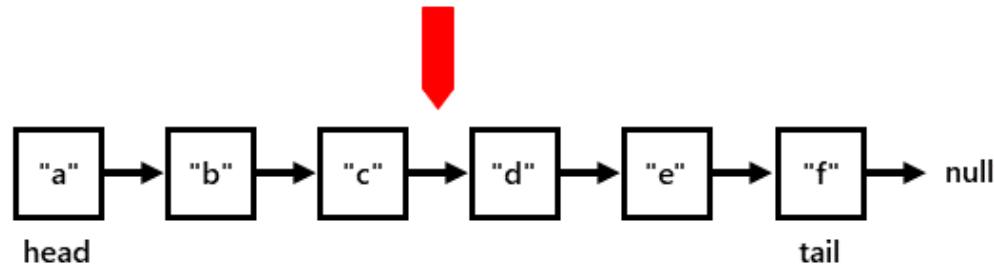


```
/**  
 * Removes the element at the specified index of this list, shifts any  
 * subsequent elements to the left (subtracts one to their indices), and  
 * returns a reference to the removed element.  
 *  
 * @param index the index of the element to remove  
 * @return the removed element  
 * @throws IndexOutOfBoundsException if the index is out of the range  
 *                                     {@code (index < 0 || index >= size())}  
 */  
  
@Override  
public E remove(int index) {  
    if (index == 0) {  
        return this.removeFront();  
    }  
    this.validate(index);  
    Node<E> n = this.moveTo(index - 1);  
    return this.removeAfter(n);  
}
```

```
/**  
 * Removes the node immediately after the specified node.  
 *  
 * @param n the node in front of the node to be removed  
 * @return the element in the removed node  
 */  
E removeAfter(Node<E> n) {  
    Node<E> toRemove = n.next;  
    Node<E> after = toRemove.next;  
    toRemove.next = null;  
    n.next = after;  
    // special case where the last element was removed  
    if (after == null) {  
        this.tail = n;  
    }  
    this.size--;  
    return toRemove.elem;  
}
```

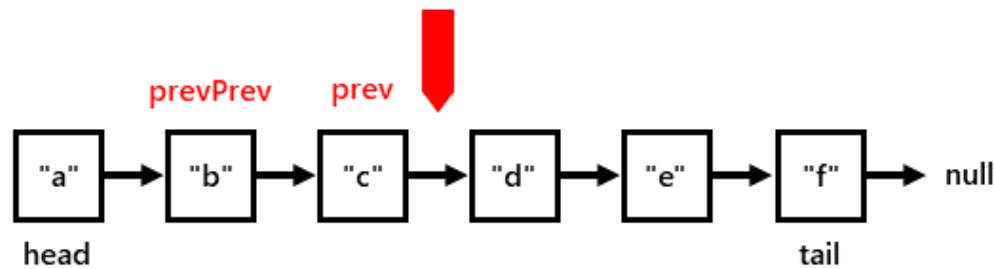
Iterator for a linked list

- ▶ recall that an iterator sits between elements
 - ▶ e.g., an iterator between the elements at indexes 2 and 3
 - ▶ the element returned by the most recent call to **next** is in the node immediately before the iterator



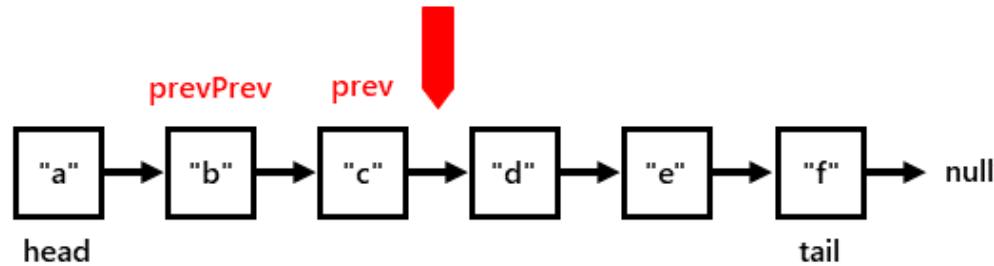
Iterator for a linked list

- we use the field **prev** to refer to the element returned by the most recent call to **next**



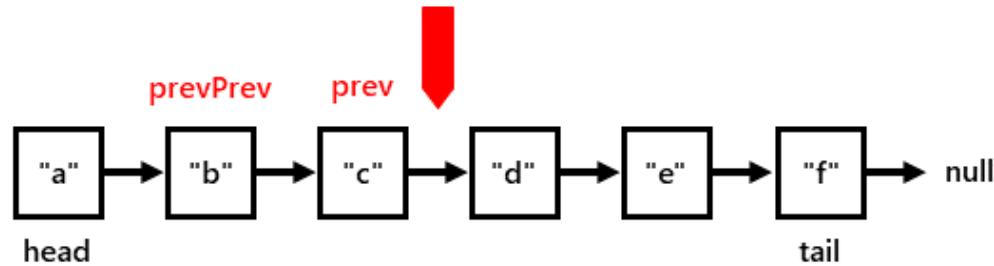
Iterator for a linked list

- ▶ **prev** is the node that will be removed when calling the iterator's **remove** method
- ▶ we require the field **prevPrev** to remove **prev** when calling the linked list method **removeAfter**



Iterator for a linked list

- ▶ **prev** is the node that will be removed when calling the iterator's **remove** method
- ▶ we require the field **prevPrev** to remove **prev** when calling the linked list method **removeAfter**



```
@Override
public Iterator<E> iterator() {
    return new LLIterator();
}

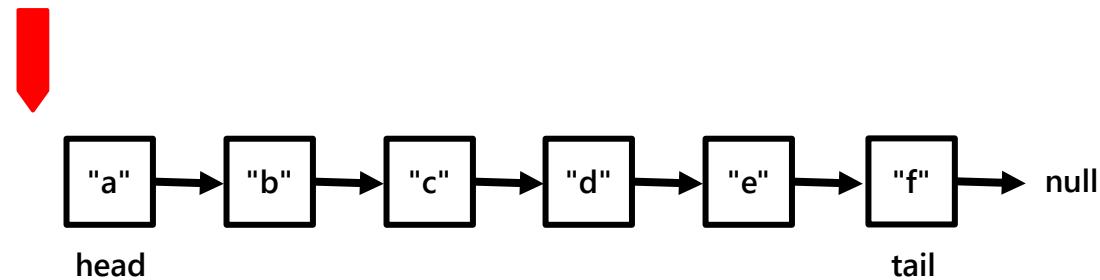
private class LLIterator implements Iterator<E> {
    /**
     * Node holding element immediately before the iterator
     */
    private Node<E> prev;

    /**
     * Node immediately before prev
     */
    private Node<E> prevPrev;

    LLIterator() {
        this.prev = new Node<>(null, SLinkedList.this.head);
        this.prevPrev = null;
    }
}
```

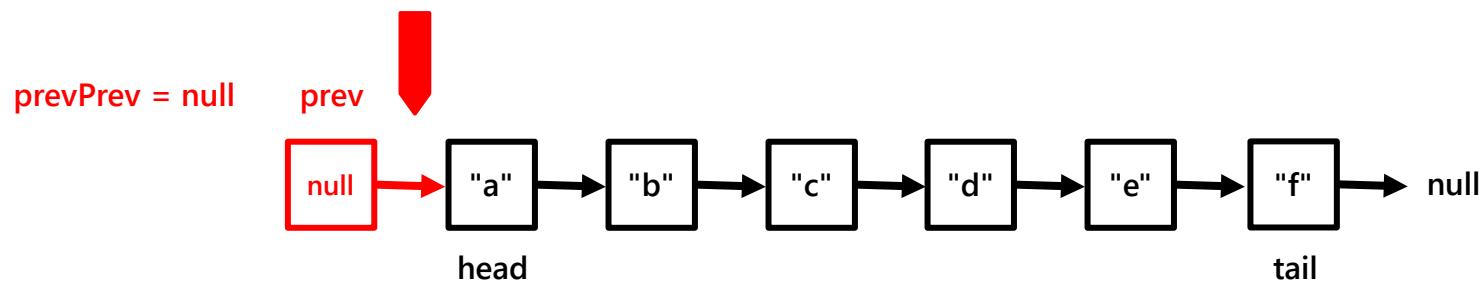
Iterator for a linked list

- ▶ at the start of an iteration, there is no node for **prev** to refer to
- ▶ to avoid dealing with special cases, it would be nice to enforce the iterator class invariant **prev != null**



Iterator for a linked list

- ▶ the iterator constructor can insert a node in front of the head node whose only purpose is to ensure that **prev** is never **null**
 - ▶ such a node is called a *sentinel* node
 - ▶ note that **prevPrev** can be **null**



```
@Override
public Iterator<E> iterator() {
    return new LLIterator();
}

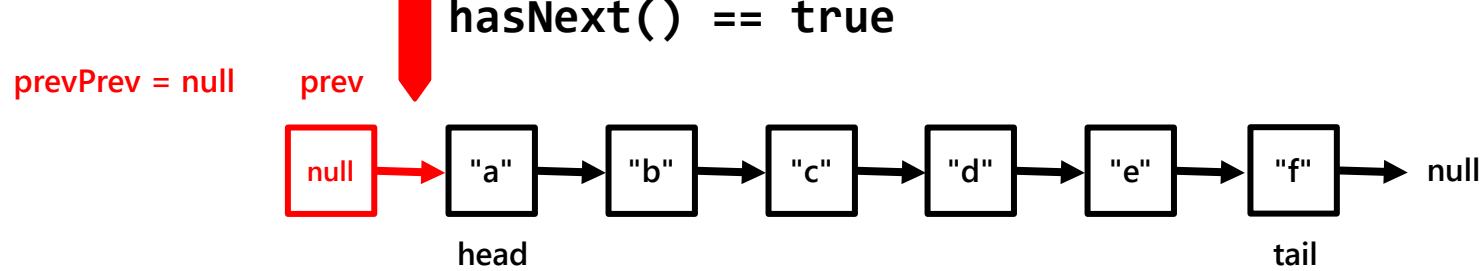
private class LLIterator implements Iterator<E> {
    /**
     * Node holding element immediately before the iterator
     */
    private Node<E> prev;

    /**
     * Node immediately before prev
     */
    private Node<E> prevPrev;

    LLIterator() {
        this.prev = new Node<>(null, SLinkedList.this.head);
        this.prevPrev = null;
    }
}
```

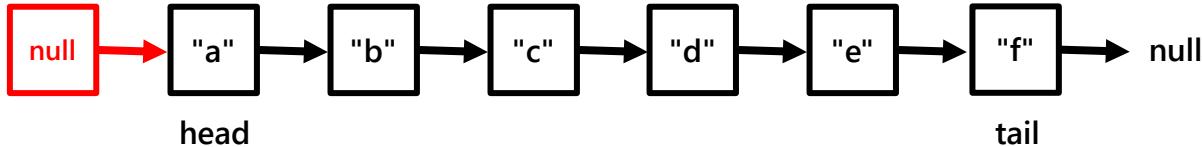
hasNext

- ▶ **hasNext** returns true if there are more elements to visit (i.e., if **prev.next != null**)



hasNext() == false

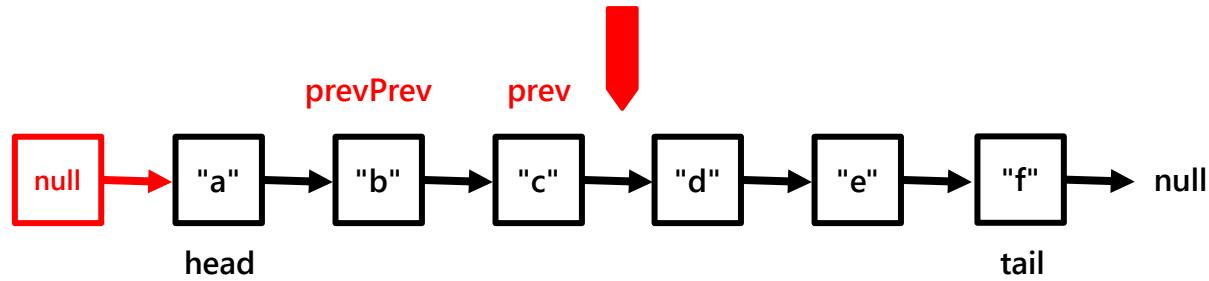
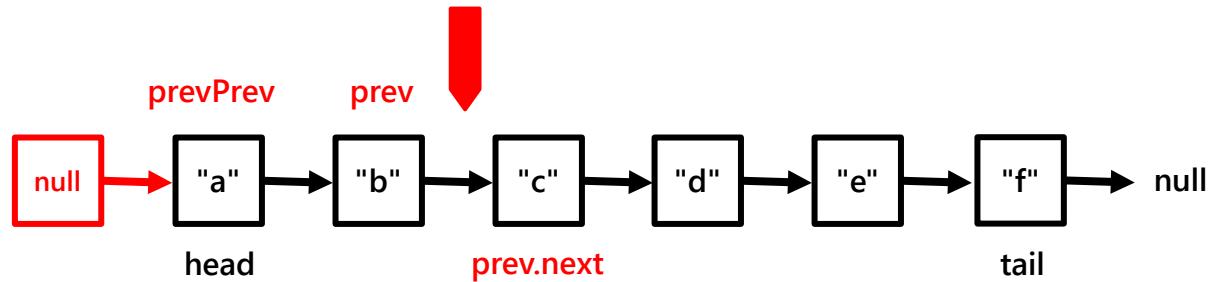
prevPrev prev



```
@Override  
public boolean hasNext() {  
    return this.prev.next != null;  
}
```

next

- ▶ **next** advances **prev** and **prevPrev** one position to the right, then returns the element in **prev**

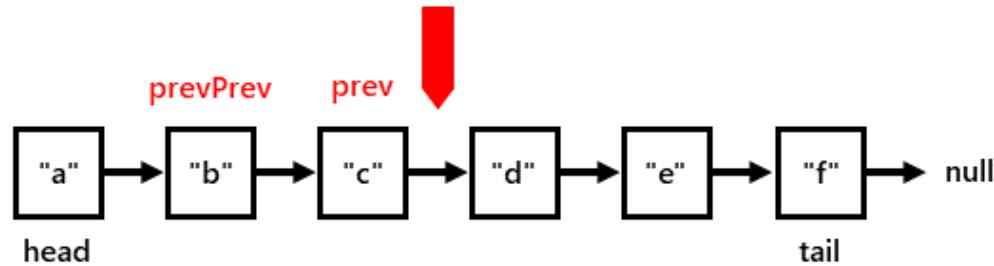


```
@Override
public boolean hasNext() {
    return this.prev.next != null;
}

@Override
public E next() {
    if (!this.hasNext()) {
        throw new NoSuchElementException();
    }
    this.prevPrev = this.prev;
    this.prev = this.prev.next;
    return this.prev.elem;
}
```

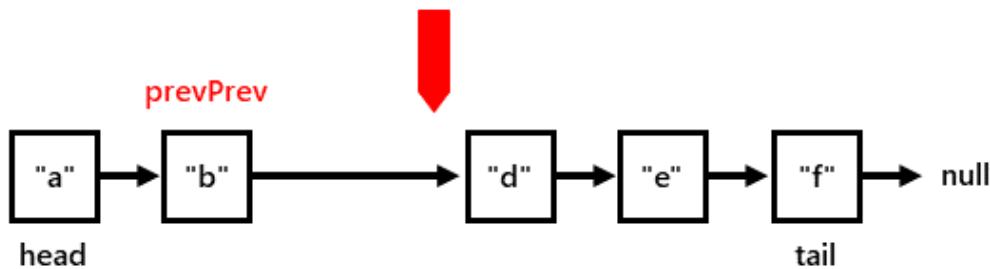
remove

- removing any element except the element in the head node can be accomplished using the **removeAfter** method
- the element to be removed is **prev**
 - call **removeAfter(prevPrev);**



remove

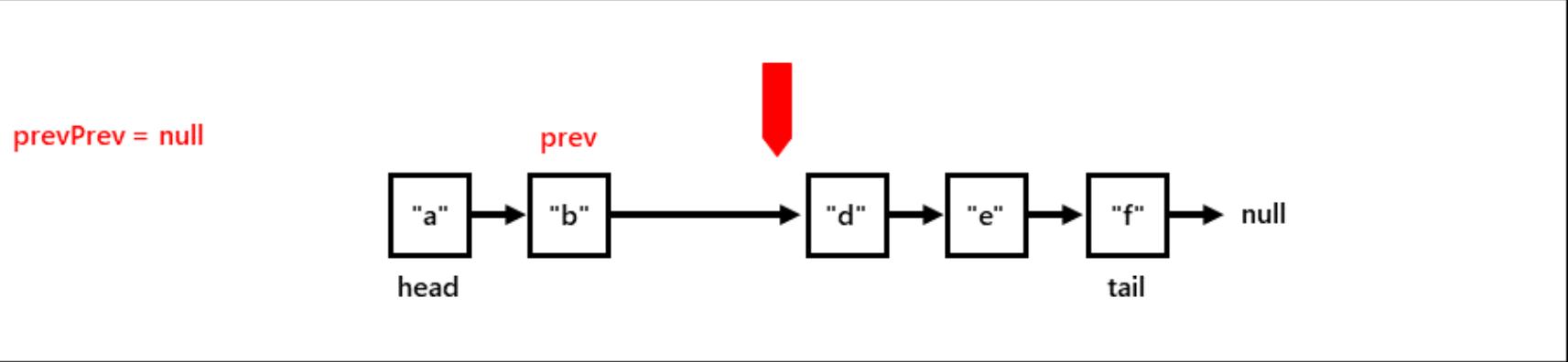
- ▶ after removing the node **prev** we get the list shown below



- ▶ **prev** should refer to the node immediately in front of the iterator, so we assign **prev = prevPrev**; and assign **prevPrev = null**;

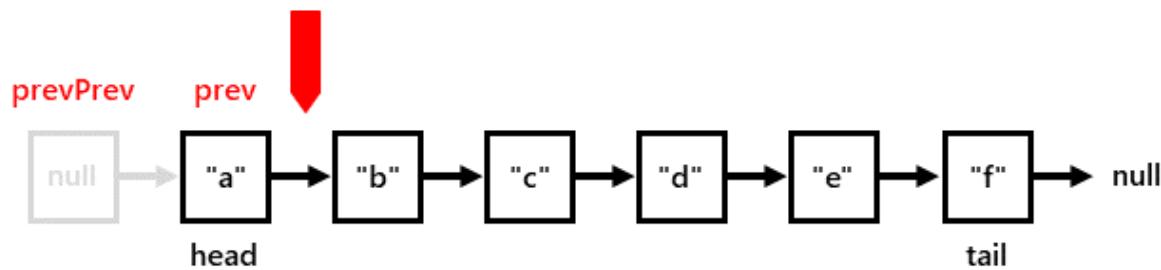
remove

- ▶ assigning `prevPrev = null`; also tells user that `remove` has been called once for the most recent call to `next`



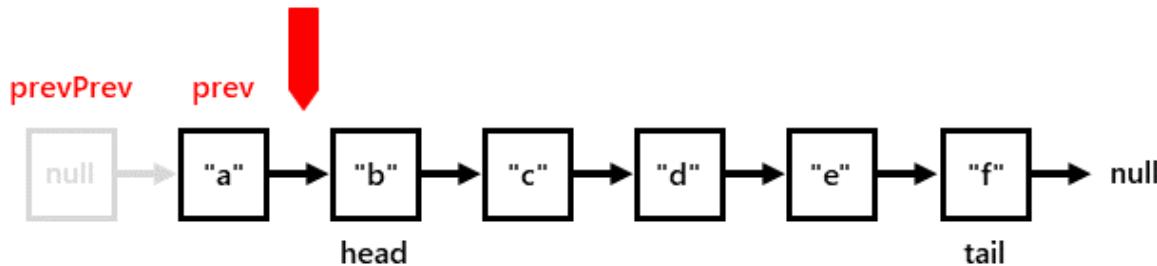
removing the first element

- if next has been called exactly once for an iterator, then the iterator has the following state:



removing the first element

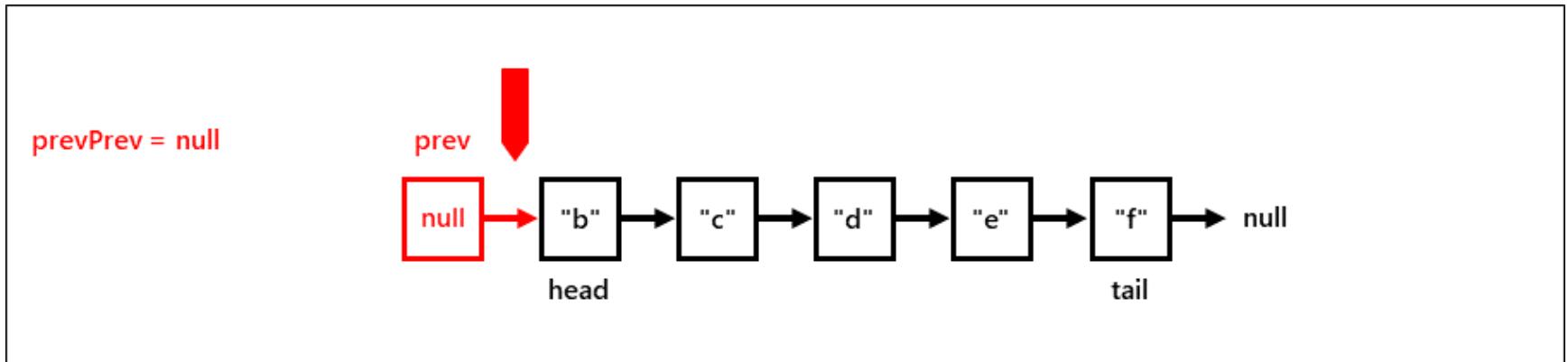
- ▶ calling **removeFront()** causes the list to adjust the field **head** so that it refers to the second element of the list:



- ▶ **prev** still refers to the old head node, and **prevPrev** still refers to the sentinel node

removing the first element

- we set the element in the node referred to by prev to null so that the element can be garbage collected, and convert the node to the sentinel node



- we assign **prevPrev = null**; to indicate that **remove** has been called

```
@Override
public void remove() {
    if (this.prevPrev == null) {
        throw new IllegalStateException();
    }
    if (this.prev == SLinkedList.this.head) {
        SLinkedList.this.removeFront();
        this.prev.elem = null;
    }
    else {
        SLinkedList.this.removeAfter(this.prevPrev);
        this.prev = this.prevPrev;
    }
    this.prevPrev = null;
}
} // end LLIIterator

} // end SLinkedList
```

Exam

- ▶ 2021/12/21, Tuesday
2:00PM - 5:00PM
Gym 3 (Bartlett) Mitchell Hall
- ▶ small set of review questions for this week's lab on onq
- ▶ more review questions+answers will appear next week

Exam: Topics

- ▶ as the course progresses, topics build on topics from earlier in the course
- ▶ therefore, you need to be familiar with all of the course material, but...
- ▶ there are no questions specifically related to the material tested on the midterm
- ▶ topics focus primarily on the notebooks from
 - ▶ Part 2 (Classes),
 - ▶ Part 3 (Relationships between classes), and
 - ▶ Part 4 (Generics and data structures)

Exam: Topics

- ▶ no need to memorize the details of class APIs
 - ▶ e.g., you will not be asked what does the method **xyz** in the class **ABC** do
 - ▶ you do need to know the basic functionality of fundamental Java classes/interfaces such as **String**, **List**, **Set**, and **Map**
- ▶ you have to write code on the exam
 - ▶ but if you find yourself writing many lines of code to answer a question then reconsider your answer
 - ▶ provide Javadoc only when asked for
- ▶ review the assignments and their solutions