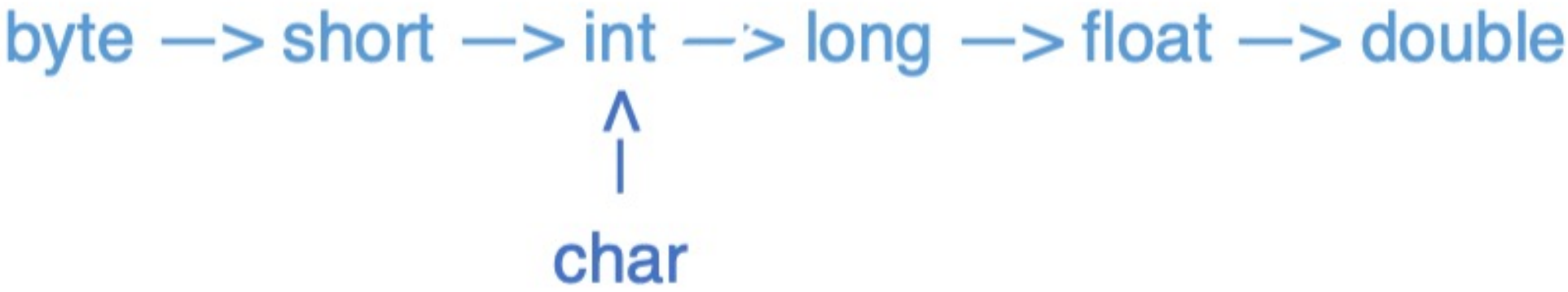
Java Types

Type	Bytes	Values
`boolean`	1	`false`, `true`
`char`	2	All unicode characters (*e.g.*, `'a'`)
`byte`	1	-2^7 to $2^7 - 1$ (-128 to 127)
`short`	2	-2^{15} to $2^{15} - 1$ (-32768 to 32767)
`int`	4	-2^{31} to $2^{31} - 1$ ($\approx \pm 2 \times 10^9$)
`long`	8	-2^{63} to $2^{63} - 1$ ($\approx \pm 10^{19}$)
`float`	4	$\approx \pm 3 \times 10^{38}$ (limited precision)
`double`	8	$\approx \pm 10^{308}$ (limited precision)



Java Types

```
4
5   public static void main(String[] args) {
6       // TODO Auto-generated method stub
7
8       boolean myboolean = true;
9       boolean myboolean = 1;
10      char mychar = 'a';
11      char mychar2 = "a";
12      char mychar3 = "char";
13      byte mybye = 15; // -128 ~ 127 < 1 byte >
14      byte mybye2 = 152;
15      short myshort = 150; // -2^16 ~ 2^16 - 1 < 2 bytes>
16      short myshort2 = (short) 150232131;
17      short myshort3 = 11212;
18      int myint = 1500000;
19      long mylong = 150000000l;
20
21      float myfloat = (float) 32.23;
22      float myfloat2 = 32.23f;
23      double mydouble = 32.33;
24
```

Pre/Post Increment/Decrement

- Pre-Increment/Decrement
- ++x/--x: returns the increment/decrement number and x becomes the increment/decrement number

```
x == 8
++x == 9
x == 9
```

```
x == 4
--x == 3
x == 3
```

- Post-Increment/Decrement
- x++/x--: returns the original number and x becomes the increment/decrement number

```
x == 8
x++ == 8
x == 9
```

```
x == 4
x-- == 4
x == 3
```

printf

- Printf is like print, but it lets you control how data is formatted
- Method from System.out.printf(format-string, args)
- Example: System.out.printf("Avarage: %5.2f", average)

Format String: %X.Y

- X is the minimum number of characters to be printed
- Y is the number of digits of the value to print after the decimal point

```
22
23        String s = "string";
24        double pi = 3.1415926535;
25        System.out.printf("\"%s\" has %d characters%n", s, s.length());
26        System.out.printf("pi to 4 places: %.4f%n", pi);
27        System.out.printf("Right>>%9.4f<<", pi);
28        System.out.printf("Left>>%-9.4f<<", pi);
29
30
31
32    }
33
34 }
35
```

Problems | Javadoc | Declaration | Console

```
<terminated> WageCalculator [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/Contents/Home/bin/java (18 Mar 2021, 4:06:3
"string" has 6 characters
pi to 4 places: 3.1416
Right>>   3.1416<<Left>>3.1416   <<
```

The final letter in a format specifier can be:

d	format an integer (no fractional part)
s	format a string (no fractional part)
c	format a character (no fractional part)
f	format a float or double
e	format a float or double in exponential notation
g	like either %f or %e, Java chooses
%	output a percent sign (no argument)
n	end the line (no argument)

- Good format for money: $%.2f

Scanner

- Get from Console: Scanner keyboard = new Scanner(System.in);
- Get from File: File text = new File("C:/temp/test.txt");
- Scanner scanner = new Scanner(text);

- .nextLine() reads the rest of the whole line as string
- .next()/ .nextInt() / .nextDouble() reads until whitespace

```
40
41          Scanner keyboard = new Scanner(System.in);
42          int n = keyboard.nextInt();
43          String s1 = keyboard.nextLine();
44          String s2 = keyboard.nextLine();
45
46          System.out.println(n);
47          System.out.println(s1);
48          System.out.println(s2);
49
50
51      }
52
53  }
54
```

Problems @ Javadoc Declaration Console

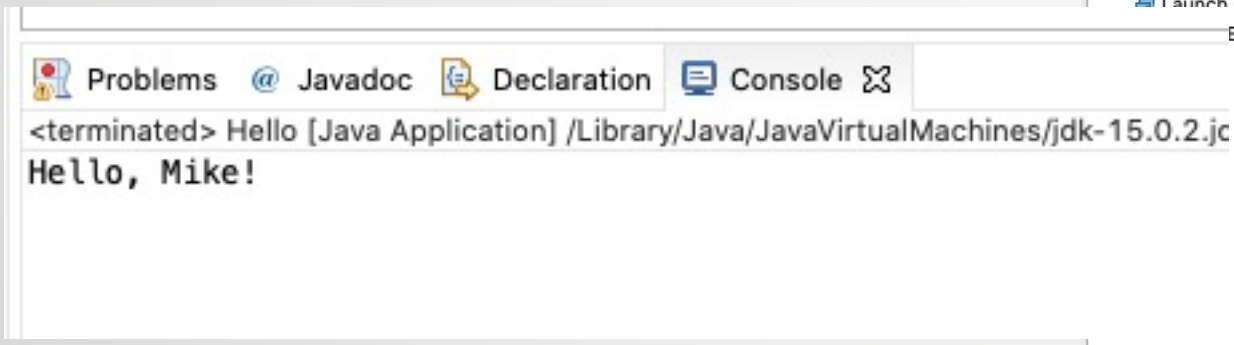
<terminated> FloatNumber [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/Contents/H

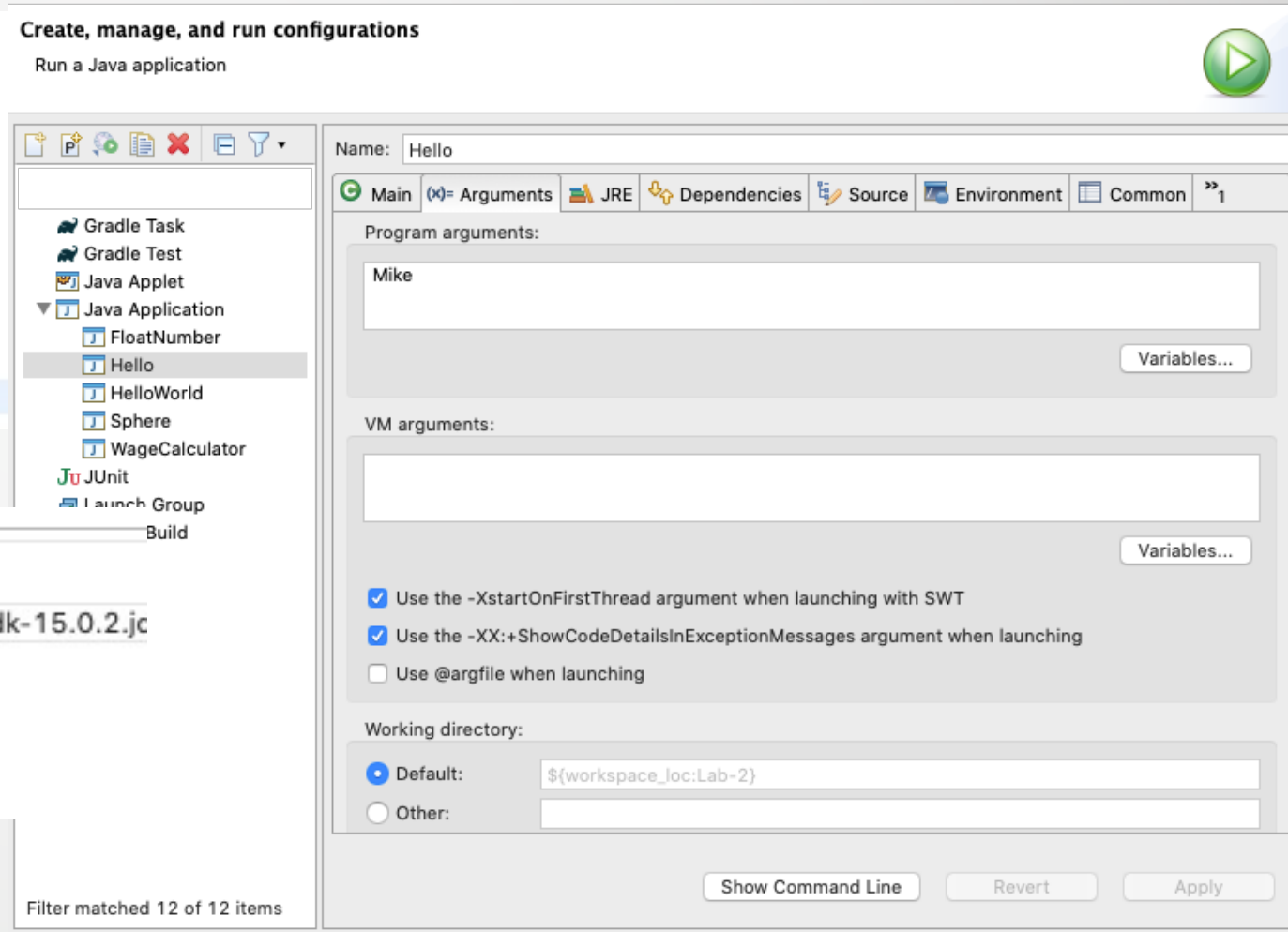
```
1
+ 2
1

+ 2
```

Command Line inputs/arguments

- First argument: args[0], second argument: args[1]
- Each of these arguments is a string (need to convert to other types for use)

```
public class Hello {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        System.out.println("Hello, " + args[0] + "!");

    }

}
```





Command Line inputs/arguments

- Convert to int
- Integer.parseInt(String)

```java
public class Hello {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        System.out.println("Hello, " + args[0] + "!");

        System.out.println("Your argumnet number is " + args[1]);

        System.out.printf("Your argumnet number is %d" , Integer.parseInt(args[1]));

    }

}
```

Conditional Operator

- Alternative for if-else statement

```
        int x = 0;

//      if (Integer.parseInt(args[1]) > 2) {
//          x = x + 2;
//      }
//      else {
//          x = x - 1;
//      }

        x = (Integer.parseInt(args[1]) > 2) ? x + 2 : x -1 ;

        System.out.print(x);


    }
```

Conditional Operator

- If - else if – else statement VS Conditional Operator

```
30          String message;
31
32          if (age < 3) {
33                message = "Hi, baby!";
34              }
35          else if (age < 18) {
36                message = "Hello!";
37              }
38          else if (age < 100) {
39                message = "Greetings!";
40              }
41          else {
42                message = "What an unusual age!";
43              }
44
45          System.out.print(message);
46
47      }
48
49  }
50
```

Problems | Javadoc | Declaration | Console

```
<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/Co
Greetings!
```

```
20  //
21          int age = 25;
22
23          String message = (age < 3) ? "Hi, baby!" :
24                (age < 18) ? "Hello!" :
25                (age < 100) ? "Greetings!" :
26                "What an unusual age!";
27
28          System.out.print(message);
```

Compound statement

- Turns multiple statements into a single statement that can be used in an if.
- A good suggestion: Always use braces!
- Always use indentation to make your code easy to read

```
29
30          String message = "Good morning";
31  //
32          if (age < 3) message = "Hi, baby!"; message += " Nice to meet you!";
33
34  //      else if (age < 18) {
35  //            message = "Hello!";
36  //          }
37  //      else if (age < 100) {
38  //            message = "Greetings!";
39  //          }
40  //      else {
41  //            message = "What an unusual age!";
42  //          }
43
44          System.out.print(message);
45  //
46      }
47
48  }
49
```

Problems @ Javadoc Declaration Console

<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/Contents/Home/bin/java (22 M

```
Good morning Nice to meet you!
```

```
29
30          String message = "Good morning";
31  //
32          if (age < 3) {
33              message = "Hi, baby!";
34              message += " Nice to meet you!";
35          }
36
37  //      else if (age < 18) {
38  //            message = "Hello!";
39  //          }
40  //      else if (age < 100) {
41  //            message = "Greetings!";
42  //          }
43  //      else {
44  //            message = "What an unusual age!";
45  //          }
46
47          System.out.print(message);
48  //
49      }
50
```

Problems @ Javadoc Declaration Console

<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines/jdk-1

```
Good morning
```

Switch

- It begins executing with the next statement and stops when it reaches a break or end of the switch.
- Always put break after each statement including default

```
48
49
50      int age = 25;
51
52      switch(age) {
53      case 24:
54          System.out.println("You are 24!");
55      case 25:
56          System.out.println("You are 25!");
57
58      case 26:
59          System.out.println("You are 26!");
60          break;
61      default:
62          System.out.println("Keep up!");
63      }
64
65  }
66  }
67
```

Problems @ Javadoc Declaration Console

```
<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/C
You are 25!
You are 26!
```

While VS do-while

- While

1. entry control loop
2. condition is checked before loop execution
3. never execute loop if condition is false
4. there is no semicolon at the end of while statement

Do-While

1. exit control loop
2. condition is checked at the end of loop
3. executes false condition at least once since condition is checked lat
4. there is semicolon at the end of while statement.

```
64
65  //  int count = 1;
66  //    while (count < 11) {
67  //        System.out.println("Count is: " + count);
68  //        count++;
69  //    }
70
71      int count = 1;
72      do {
73          System.out.println("Count is: " + count);
74          count++;
75      } while (count < 11);
76
77  }
78  }
79
80
```

Problems @ Javadoc Declaration Console

<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/Contents

```
Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10
```

While VS do-while

```
70
71      int count = 11;
72      do {
73          System.out.println("Count is: " + count);
74          count++;
75      } while (count < 11);
76
77  }
78  }
79
80
```

Problems @ Javadoc Declaration Console

<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines

```
Count is: 11
```

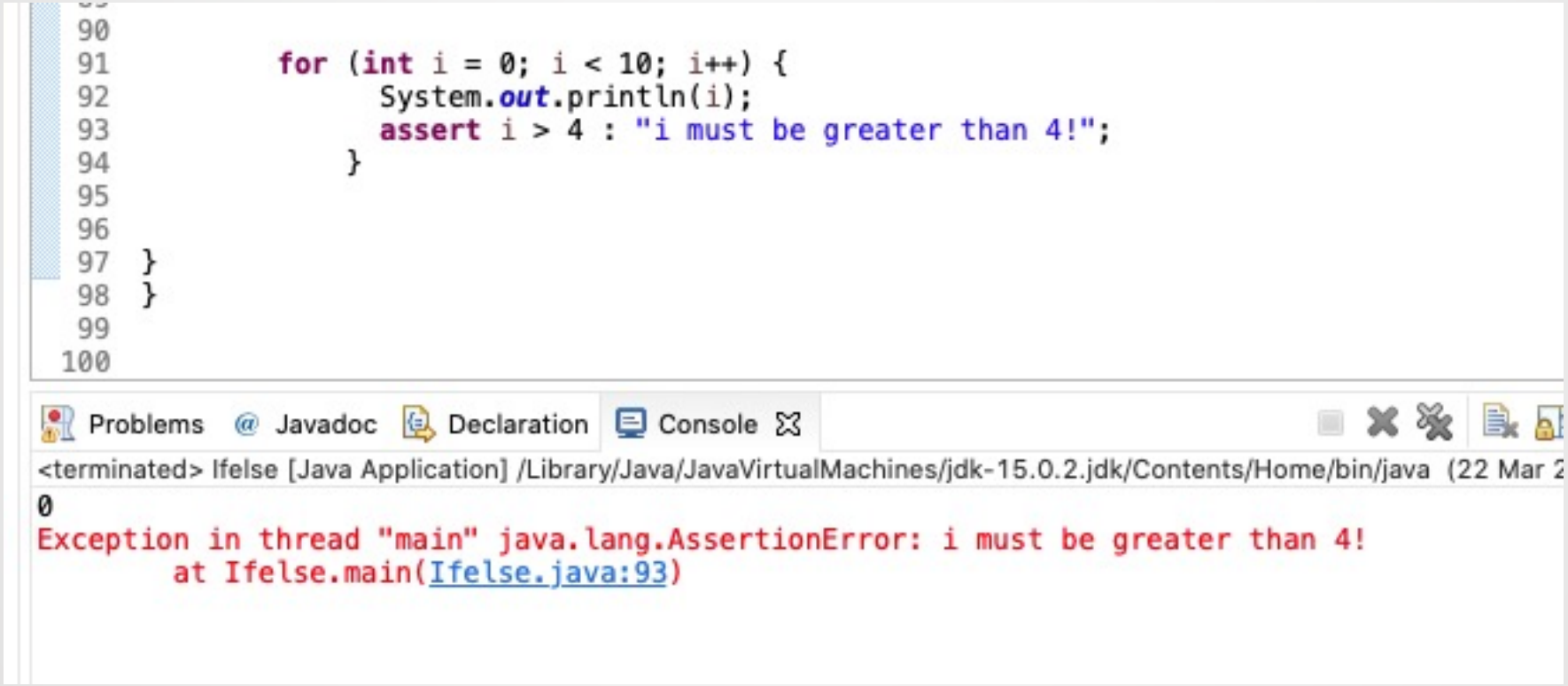
```
64
65      int count = 11;
66      while (count < 11) {
67          System.out.println("Count is: " + count);
68          count++;
69      }
70
71  //      int count = 11;
72  //      do {
73  //          System.out.println("Count is: " + count);
74  //          count++;
75  //      } while (count < 11);
76
77  }
78  }
79
80
```

Problems @ Javadoc Declaration Console

<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/Contents/Hom

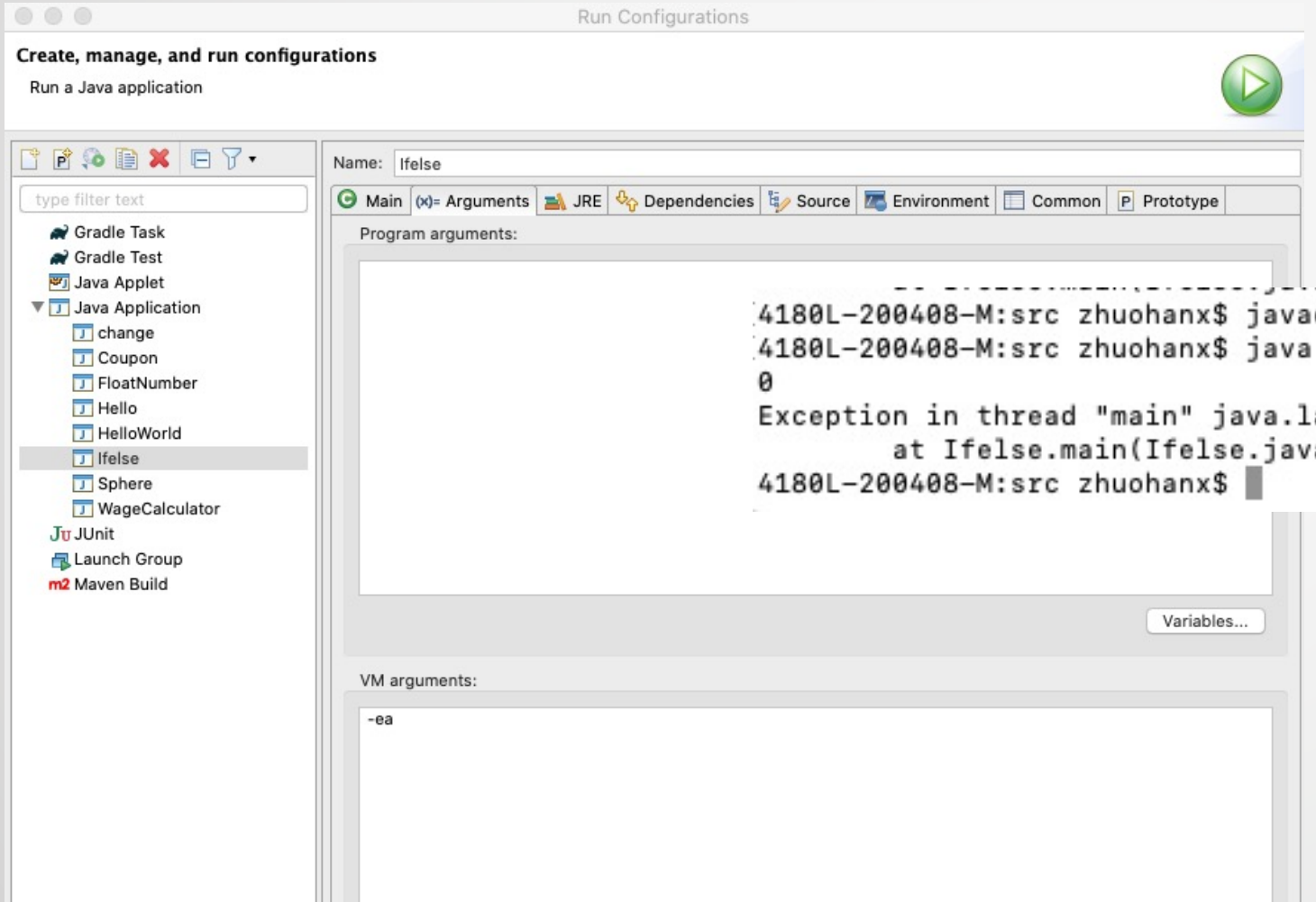
Assert

- Check for some must meet condition!
- Not common use
- Enable Asset in Eclipse



Assert

- Check for some must meet condition!
- Not common use
- Enable Asset in Eclipse or Terminal




```
4180L-200408-M:src zhuohanx$ javac Ifelse.java
4180L-200408-M:src zhuohanx$ java -ea Ifelse
0
Exception in thread "main" java.lang.AssertionError: i must be greater than 4!
        at Ifelse.main(Ifelse.java:88)
4180L-200408-M:src zhuohanx$
```

Continue VS Break

- Continue passes one iteration; break breaks out of the whole loop

```
82
83          for (int i = 0; i < 10; i++) {
84                if (i == 4) {
85                  continue;
86                }
87                System.out.println(i);
88              }
89
90
91
92
93  }
94  }
95
96
```

Problems @ Javadoc Declaration Console

<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.0.2.jdk/Conte

```
0
1
2
3
5
6
7
8
9
```

```
89
90          for (int i = 0; i < 10; i++) {
91                if (i == 4) {
92                  break;
93                }
94                System.out.println(i);
95              }
96
97
98  }
99  }
100
101
```

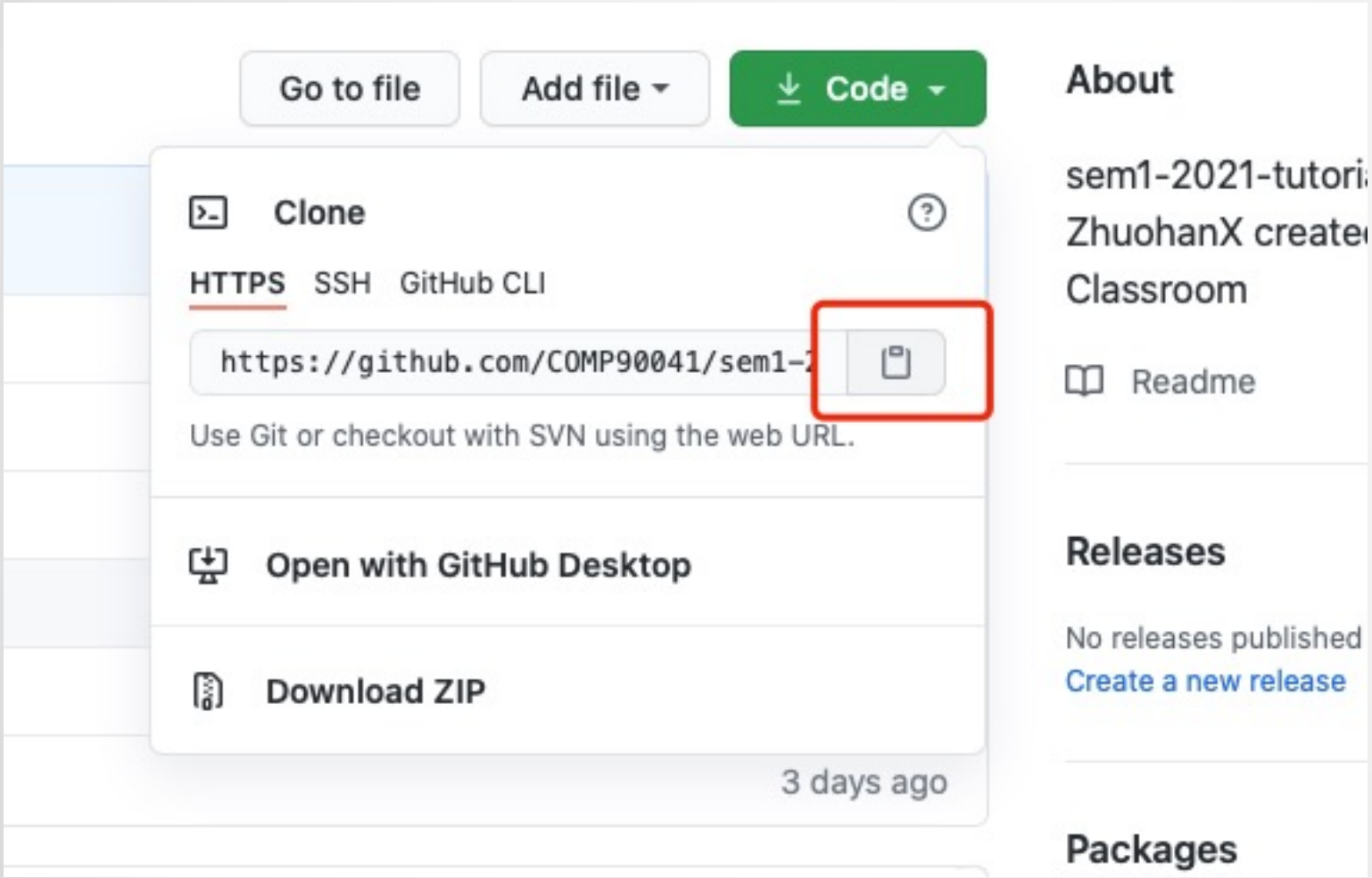
Problems @ Javadoc Declaration Console

<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines/jdk-15.0

```
0
1
2
3
```

Git Clone a repository

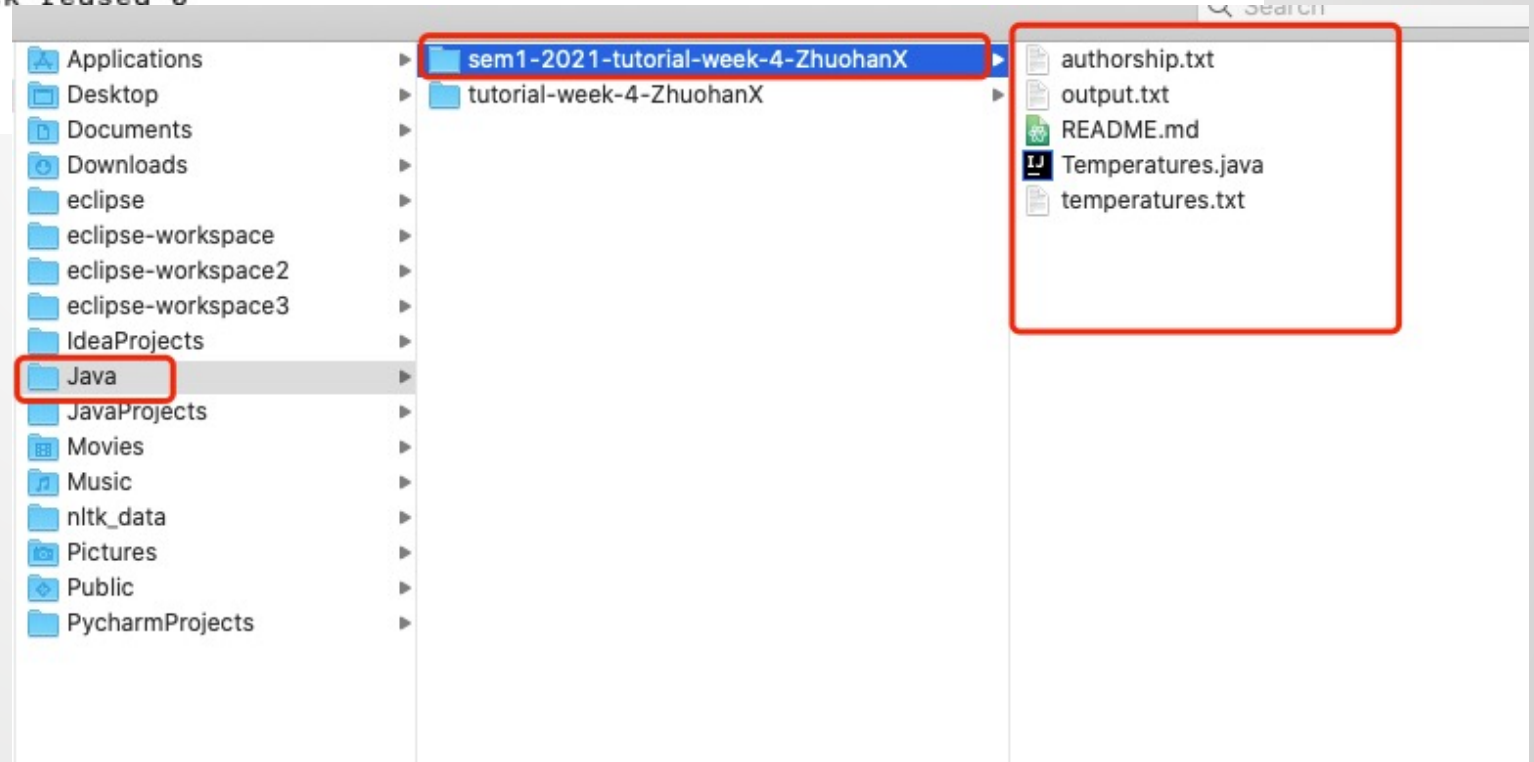
- Copy the link!



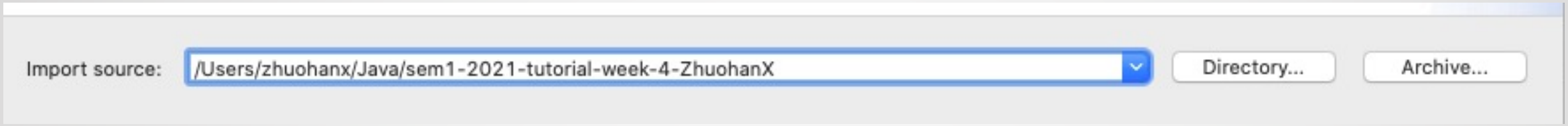
Git Clone a repository

- Open a terminal at the suitable location and use git clone (link)

```
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
4180L-200408-M:Java zhuohanx$ git clone https://github.com/COMP90041/sem1-2021-tutorial-week-4-ZhuohanX.git
Cloning into 'sem1-2021-tutorial-week-4-ZhuohanX'...
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 8 (delta 0), reused 6 (delta 0), pack-reused 0
Unpacking objects: 100% (8/8), done.
4180L-200408-M:Java zhuohanx$
```

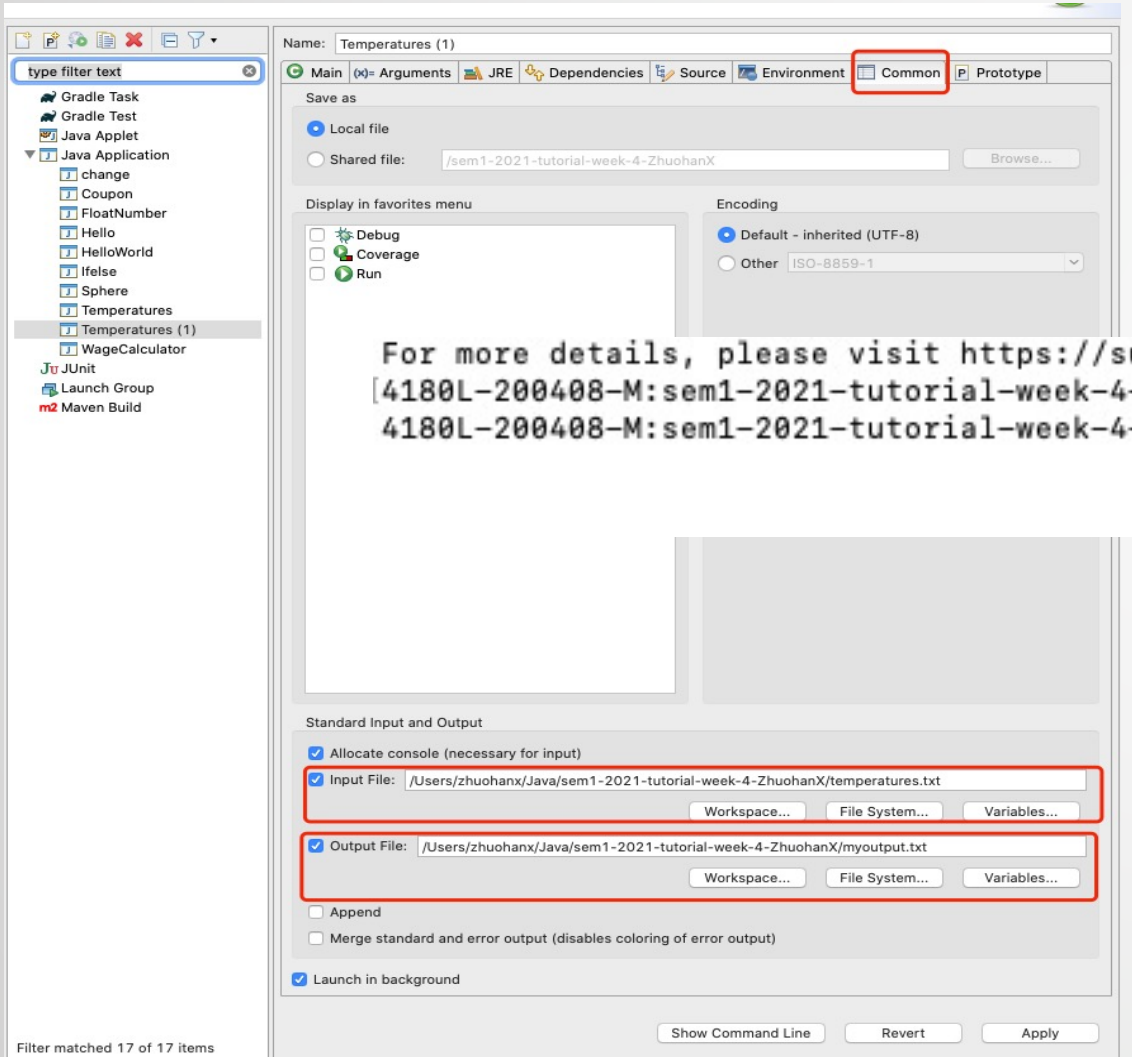


Import from File System and Edit Your Code



Run code with file input and output

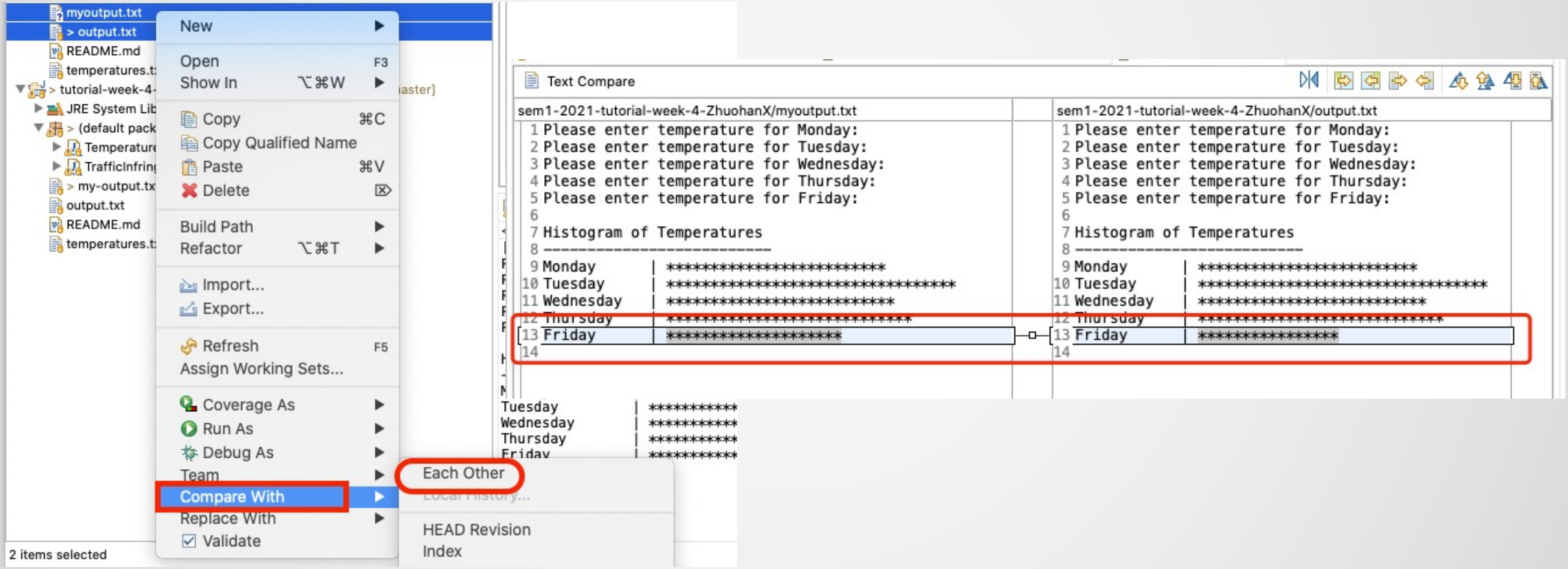
- Eclipse: Run configuration > Common > Input File and Output File.
- Terminal : Java (Class) < (input file) > (output file)
- Command can be separated (use only input or output)




```
For more details, please visit https://support.apple.com/kb/HT208050.
4180L-200408-M:sem1-2021-tutorial-week-4-ZhuohanX zhuohanx$ java Temperatures < temperatures.txt > myoutput.txt
4180L-200408-M:sem1-2021-tutorial-week-4-ZhuohanX zhuohanx$
```

Compare the Differences

- Eclipse: Select two files and use compare with each other



Compare the Differences

- Terminal command: sdiff and diff

```
4180L-200408-M:sem1-2021-tutorial-week-4-ZhuohanX zhuohanx$ diff myoutput.txt output.txt
13c13
< Friday                | **********
---
> Friday                | **********
4180L-200408-M:sem1-2021-tutorial-week-4-ZhuohanX zhuohanx$ sdiff myoutput.txt output.txt
Please enter temperature for Monday:                    Please enter temperature for Monday:
Please enter temperature for Tuesday:                   Please enter temperature for Tuesday:
Please enter temperature for Wednesday:                 Please enter temperature for Wednesday:
Please enter temperature for Thursday:                  Please enter temperature for Thursday:
Please enter temperature for Friday:                    Please enter temperature for Friday:

Histogram of Temperatures                               Histogram of Temperatures
-------------------------                               -------------------------
Monday          | **************************            Monday          | **************************
Tuesday         | **********************************    Tuesday         | **********************************
Wednesday       | **************************            Wednesday       | **************************
Thursday        | ****************************          Thursday        | ****************************
Friday          | **********                          | Friday          | **********
4180L-200408-M:sem1-2021-tutorial-week-4-ZhuohanX zhuohanx$
```

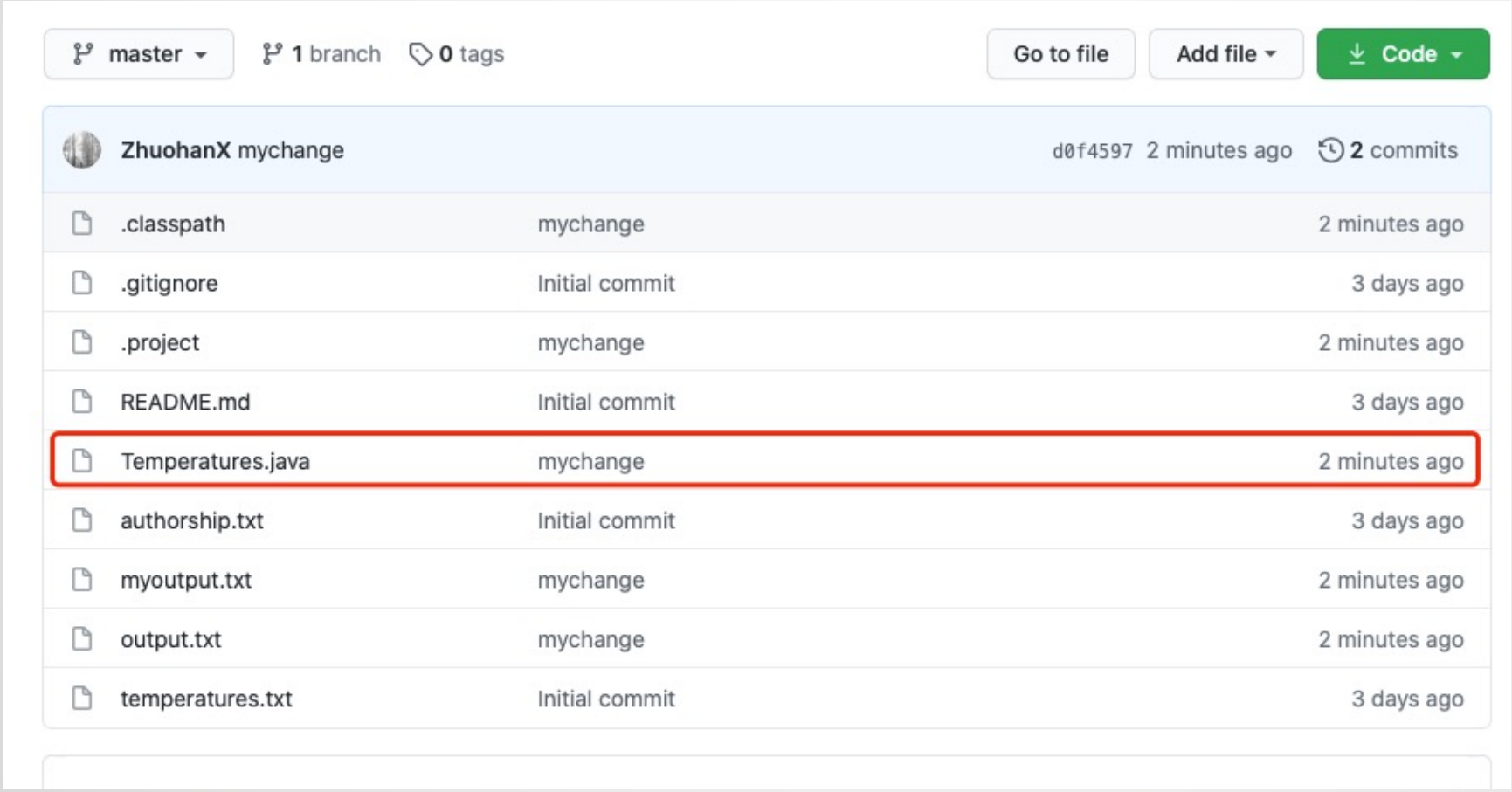
Commit code change and push

- Add all your changed code files (. Means all)
- Commit with commit message (for yourself to check)
- Push to the remote repostitory

```
For more details, please visit https://support.apple.com/kb/HT208050.
4180L-200408-M:sem1-2021-tutorial-week-4-ZhuohanX zhuohanx$ git add .
4180L-200408-M:sem1-2021-tutorial-week-4-ZhuohanX zhuohanx$ git commit -m "mychange"
[master d0f4597] mychange
 5 files changed, 86 insertions(+), 6 deletions(-)
 create mode 100644 .classpath
 create mode 100644 .project
 create mode 100644 myoutput.txt
4180L-200408-M:sem1-2021-tutorial-week-4-ZhuohanX zhuohanx$ git push
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 1.49 KiB | 1.49 MiB/s, done.
Total 6 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/COMP90041/sem1-2021-tutorial-week-4-ZhuohanX.git
   813ef9e..d0f4597  master -> master
4180L-200408-M:sem1-2021-tutorial-week-4-ZhuohanX zhuohanx$
```

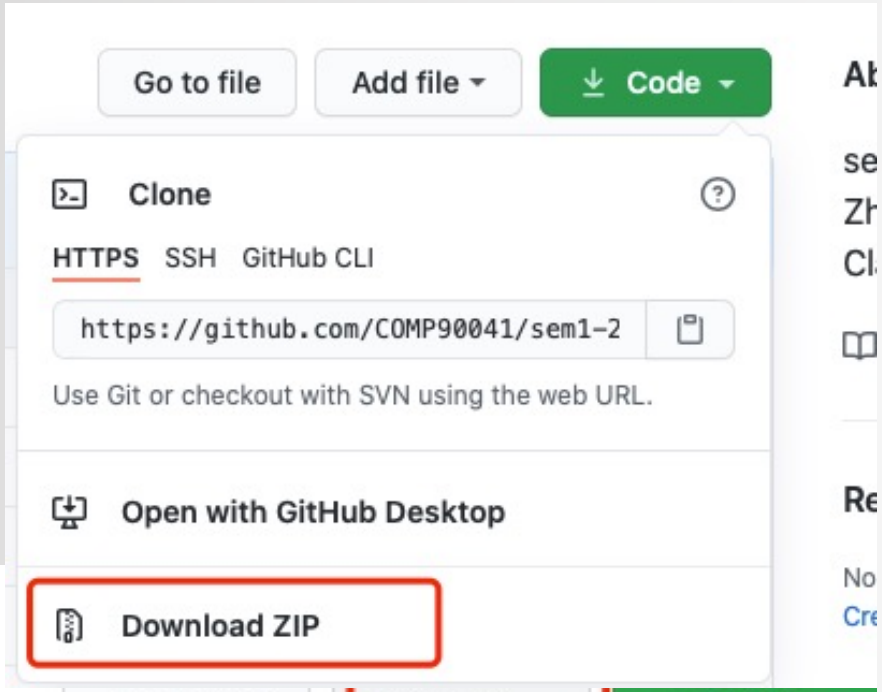
Commit code change and push

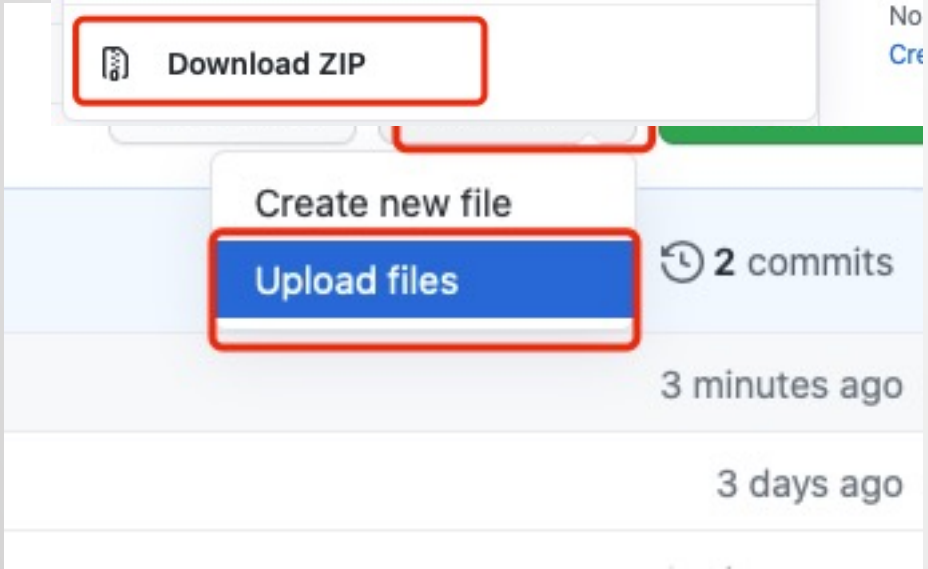
- Updated file shows message and time!

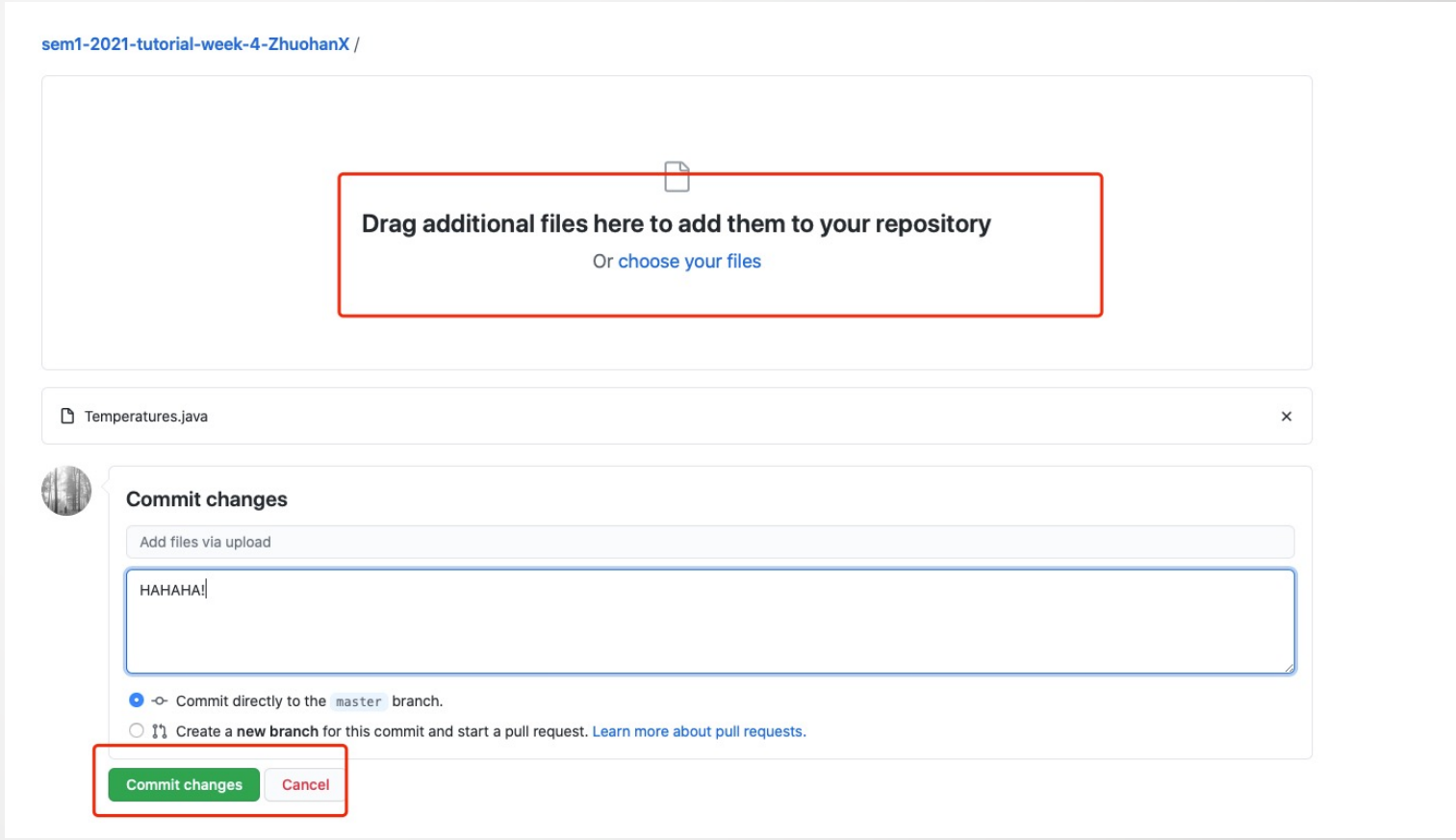


Do everything on Website

- Download Code and Edit and drag your files and commit on webpage.







Classes and Methods

- A Java program consists of objects from various classes interacting with one another
- Every program is a class
- Class is a type that you can declare variables of a class type
- Object or instance of the class is the value of a class type
- Car mycar = new Car();
- Object has both data and actions (methods).
- All objects of the same class might have different data value, but the same data type and methods.

- A primitive type value is a single piece of data
- A class type object can have multiple pieces of data and methods.

```
1
2  public class AppointmentDate {
3
4
5
6      private int year = 2021;
7      private int month = 2;
8      private int day = 10;
9
10
11     public AppointmentDate(){
12         month = 1;
13         day = 1;
14         year = 2020;
15     }
16
17 }
18
```

Methods

- Method consists of a heading and a method body
- Methods are invoked using the name of the calling object or class name and the method name
- Class.method(); or classvar.method();
- Method can compute and return a type value or just perform an action and no return (void method)
- A program in Java is a class that contains a main method (a void method)

- public static void main(String[] args)

return

- The body of a method that returns a value must contain one or more return statements
- A return statement specifies the value returned and ends the method invocation: return Expressions;
- A void method need not contain a return statement but can use return to end the code.

```
public static void number() {
    for (int j = 0 ; j < 10 ; j ++) {

        if (j == 4) {
            return;
          }
        System.out.println(j);
    }

}

public static void main(String[] args) {

    number();
```

```
<terminated> Ifelse [Java Application] /Library/Java/JavaVirtualMachines/
0
1
2
3
```

Use returned value

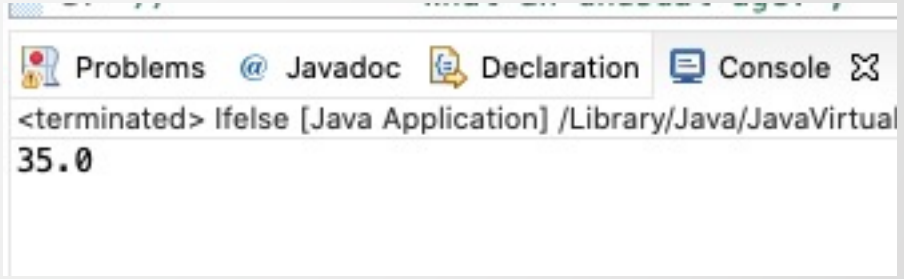
- A invocation of a method that returns a value can be used anyplace that a value of the RetrunedType can be used.
- An invocation of a method that returns a value of type boolean returns either true or false, usually used in if-else statement or while loops.

```
public static void number() {
    for (int j = 0 ; j < 10 ; j ++) {

        if (checkvalue(j)) {
            return;
          }
        System.out.println(j);
    }

}


public static boolean checkvalue(int value) {

    if (value > 4 && value < 10) {
        return true;
    }


    return false;
}
```

Parameter and argument

- A parameter list provides a description of the data required by a method
- It indicates the number and types of data pieces needed, the order they must be given and the local name for these pieces as used in the method
- Argument is the appropriate values passed to the method (Actual parameters)
- The number and order of arguments must exactly match that of the parameter list
- If types do not match, Java will try to make an automatic type conversion
- call-by-value: value of each argument is passed into the method parameter not name

```
public static double number(int a, int b, double c) {
    return (a + b) *c;

}

public static void main(String[] args) {

    int d = 3, e=4, f =5;
    System.out.print(number(d,e,f));
```



this

- this parameter is a hidden parameter
- this must be used if local variables have the same name with the instance variables, otherwise all variables will be interpreted as local

```
private int year = 2021;
private int month = 2;
private int day = 10;


public void setDate(int year, int month, int day){
    if (isValidDate(year, month, day) ) {
        this.year = year;
        this.month = month;
        this.day = day;
    }
}
```

equal and toString

- Java expects certain methods such as equal and toString
- equal is to compare if the two objects of the class type are the same
- toString can be used to display the value of the object

```
public boolean equal(AppointmentDate otherDate) {
    if (otherDate.year == year && otherDate.month == month
            && otherDate.day == day) {
        return true;
    }
    return false;
}

public String toString() {
    return (year + "-" + month + "-" + day);
}
```

Testing methods

- Driver program: A program with only purpose to test a method
- Bottom-up testing: For some method that invokes other methods, First test all methods it invokes and then test such method

```
public boolean equal(AppointmentDate otherDate) {
    if (otherDate.year == year && otherDate.month == month
            && otherDate.day == day) {
        return true;
    }
    return false;
}

public String toString() {
    return (year + "-" + month + "-" + day);
}
```

Information hiding and Encapsulation

- Information hiding is the practice of separating how to use a class from the details of its implementation
- Abstraction is another term used to express the concept of discarding details in order to avoid information overload
- Encapsulation means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
- The API or application programming interface for a class is a description of how to use the class
- An ADT or abstract data type is a data type that is written using good information-hiding techniques

Interface

```java
public interface Animal {

    public void animalSound(); // interface method (does not have a body)
    public void sleep(); // interface method (does not have a body)

}
```

```java
public class Pig implements Animal {

    @Override
    public void animalSound() {
        // TODO Auto-generated method stub
        System.out.println("The pig says: wee wee");
    }

    @Override
    public void sleep() {
        // TODO Auto-generated method stub
        System.out.println("Zzz");
    }

}
```

public and private

- public means that there are no restrictions on where an instance variable or method can be used
- private means that an instance variable or method cannot be accessed by name outside of the class
- It is considered good programming practice to make all instance variables private
- Most methods are public, and thus provide controlled access to the object
- Usually, methods are private only if used as helping methods for other methods in the class

Accessor and Mutator

- Accessor methods allow the programmer to obtain the value of an object's instance variables (getter)
- Mutator methods allow the programmer to change the value of an object's instance variables in a controlled manner (setter)

```
public int getYear() {return year;}

public int getMonth() {return month;}

public int getDay() {return day;}
```

```
public void setDate(int year, int month, int day){
    if (isValidDate(year, month, day) ) {
        this.year = year;
        this.month = month;
        this.day = day;
    }
}
public void setMonth(int month){
    this.month = month;
}
```

Overloading

- Overloading is when two or more methods in the same class have the same method name
- A signature consists of the name of a method together with its parameter list (not include the returned type)
- To be valid, any two definitions of the method name must have different signatures
- Java does not permit methods with the same name and different return types in the same class

Overloading

```
public void setDate(int aDay,
            int aMonth, int aYear)
{
  day = aDay;
  month = aMonth;
  year = aYear;
}

public void setDate(int aDay,
            String aMonth, int aYear)
{
  day = aDay;
  month = convertMonth(aMonth);
  year = aYear;
}
```

```
//helper methods
private int convertMonth(String aMonth)
{
  if (aMonth.equalsIgnoreCase("January"))
    return 1;
  else if (aMonth.equalsIgnoreCase("February"))
    return 2;
  else if (aMonth.equalsIgnoreCase("March"))
    return 3;
  else if (aMonth.equalsIgnoreCase("April"))
    return 4;
  else if (aMonth.equalsIgnoreCase("May"))
    return 5;
  else if (aMonth.equalsIgnoreCase("June"))
    return 6;
  else if (aMonth.equalsIgnoreCase("July"))
    return 7;
  else if (aMonth.equalsIgnoreCase("August"))
    return 8;
  else if (aMonth.equalsIgnoreCase("September"))
    return 9;
  else if (aMonth.equalsIgnoreCase("October"))
    return 10;
  else if (aMonth.equalsIgnoreCase("November"))
    return 11;
  else if (aMonth.equalsIgnoreCase("December"))
    return 12;
  else
  {
```

Constructor

- Constructor is a special kind of method that is designed to initialize the instance variables for an object
- A constructor must have the same name as the class
- A constructor has no type returned, not even void
- Constructors are typically overloaded
- A constructor is called when an object of the class is created using new

Constructor

```
//constructors
public Date()
{
  day = 1;
  month = 1;
  year = 1000;
}

public Date(int aDay, int aMonth, int aYear)
{
  day = aDay;
  month = aMonth;
  year = aYear;
}

public Date(int aDay, String aMonth, int aYear)
{
  day = aDay;
  month = convertMonth(aMonth);
  year = aYear;
}
```

static methods and variables

- A static method can be used without a calling object, need to use the keyword static
- public static returnedType MethodName(parameters…)
- A static method is invoked using the classname.methodname()
- A static method has no this inside the method, so it cannot refer to an instance variable of the class or invoke a non-static method of the class, but it can invoke another static method

- A static variable is a variable that belongs to the class as a whole, not just to one object.
- Only one copy of a static variable per class, unlike instance variable where each object has its own copy
- All objects of the class can read and change a static variable
- Static method can access a static variable
- Math class has a lot methods you can call using Math.methodnames()

Wrapper classes for primitive types

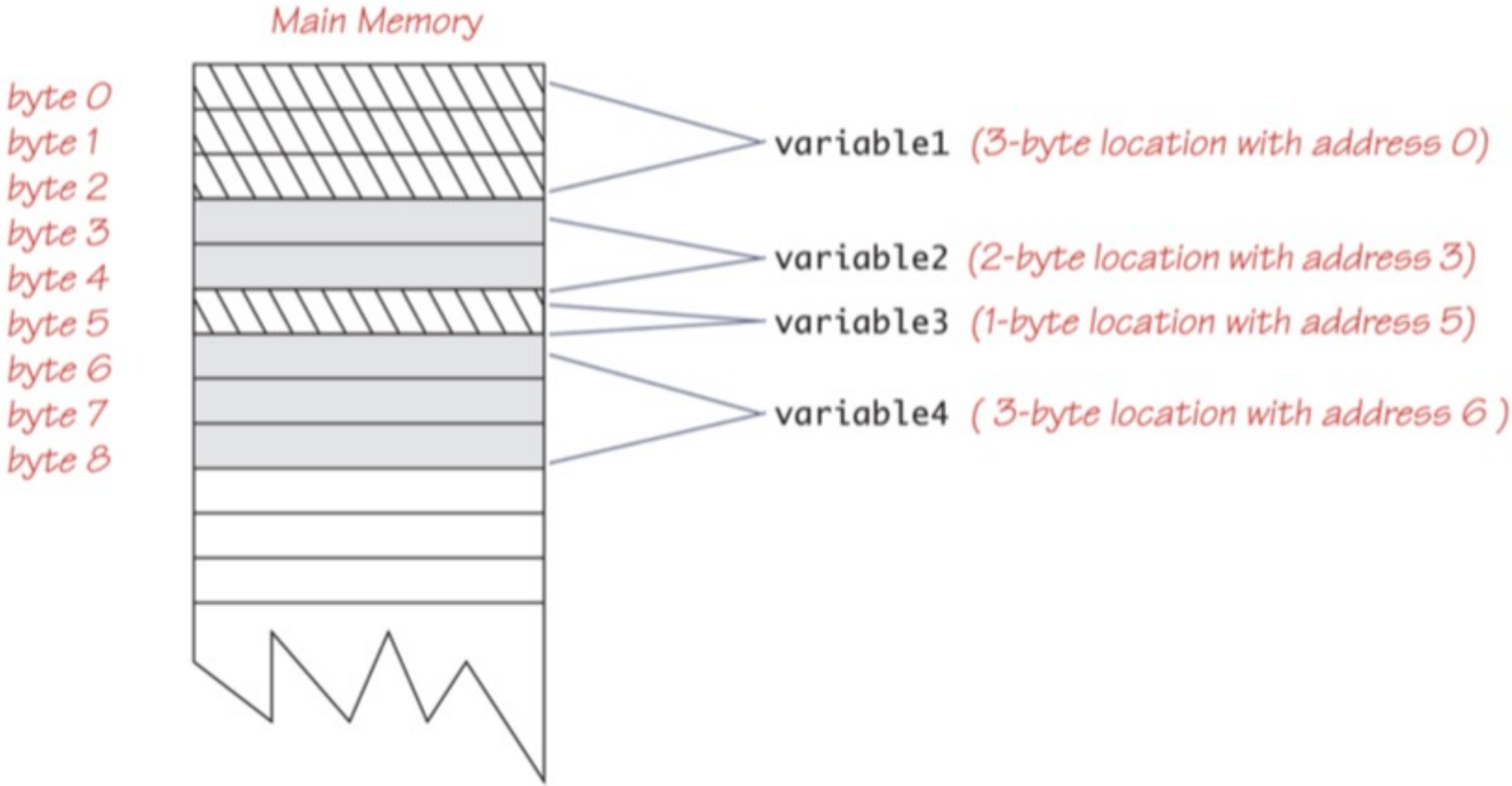
- Wrapper classes provide a class type corresponding to each of the primitive types and makes it possible to have class types that behave somewhat like primitive types.
- Boxing: go from a primitive type to an object of its wrapper class
- Unboxing: go from an object of wrapper class to the corresponding value of a primitive type
- Boxing: Integer integerObject = new Integer(2);
- Unboxing: int i = integerObject.intValue();

Wrapper classes for primitive types

- Wrapper classes provide a class type corresponding to each of the primitive types and makes it possible to have class types that behave somewhat like primitive types.
- Boxing: go from a primitive type to an object of its wrapper class
- Unboxing: go from an object of wrapper class to the corresponding value of a primitive type
- Boxing: Integer integerObject = new Integer(2);
- Unboxing: int i = integerObject.intValue();

Variables and Memory

- Second memory is used to hold files for “permanent” storage
- Main memory (primary memory) is used by the computer when running a program
- Values in a program are kept in main memory
- Main memory consists of a long list of numbered locations called bytes
- Address: the number that identifies a byte
- Values of most data types require more than one byte of storage and the
- Address of the first byte of this memory location is used as the address for the data
- When the variable is a primitive type, the value of the variable is stored in the memory location, each primitive type always require the same amount of memory to store its values
- When the variable is a class type, only the memory address (or reference) where the object is located is stored in the memory location


Main Memory
byte 0
byte 1
byte 2
byte 3
byte 4
byte 5
byte 6
byte 7
byte 8
variable1 (3-byte location with address 0)
variable2 (2-byte location with address 3)
variable3 (1-byte location with address 5)
variable4 (3-byte location with address 6)

Class Assignment and Parameters

- Two reference variables can contain the same reference, assignment sets the reference to the same object
- variable1 = variable2;

- A method cannot change the value of a variable of a primitive type that is an argument to the method
- In contrast, a method can change the values of the instance variables of a class type that is an argument to the method.

- == operator checks that two class type variables have the same memory address
- equals checks two class type variables have the same data values (often write by yourself)
- null is a ”placeholder” for a reference that does not name any memory location, it is like a memory address, use == or != not equals

```java
public class ToyClass {

    private String name;
    private int number;

    public ToyClass(String name, int number) {
        this.name = name;
        this.number = number;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    public int getNumber() {return number;}

    public String toString() {
        return name + "    " + number;
    }

}
```

Problems @ Javadoc Declaration Console

<terminated> ToyClassDemo [Java Application] /Library/Java/Ja

```
Mike   20
Mike   20
Taylor   20
Taylor   20
5
4
21
20
21
21
```

```java
public class ToyClassDemo {

    public static int addone(int num) {
        num += 1;
        return num;
    }

    public static int addone(ToyClass num) {
        num.setNumber(num.getNumber()+1);
        return num.getNumber();
    }

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        ToyClass sampleVariable1 = new ToyClass("Mike", 20);
        System.out.println(sampleVariable1);

        ToyClass sampleVariable2 = sampleVariable1;

        System.out.println(sampleVariable2);

        sampleVariable2.setName("Taylor");
        System.out.println(sampleVariable2);
        System.out.println(sampleVariable1);

        int n = 4;

        System.out.println(addone(n));

        System.out.println(n);

        System.out.println(addone(sampleVariable1.getNumber()));
        System.out.println(sampleVariable1.getNumber());

        System.out.println(addone(sampleVariable1));
        System.out.println(sampleVariable1.getNumber());

    }

}
```

Immutable and mutable Class

- A class that contains no methods (other than constructors) that change any of the data in an object of the class is called an immutable class (String class is an immutable class)
- Objects of such classes are called immutable objects
- A class that contains public mutator methods or other public methods that can change the data in its objects is called a mutable class
- Objects of which are mutable objects

Deep and Shallow Copy

- A copy constructor is a constructor with a single argument of the same type as the class
- A deep copy of an object is a copy that with one exception has no references in common with the original. Exception: References to immutable objects
- Any copy that is not a deep copy is called a shallow copy

- Privacy leak: Shallow Copy without create a new instance variable for mutable class or return a mutable object can cause privacy leak. The values can be changed by others

Arrays

- An array that behaves like this collection of variables, all of type double, can be created using one statement as follows:
- double[] score = new double[5];
- Use index score[0], score[1], score[2], score[3], score[4] to get the number 1, 2, 3, 4, 5 of the element.

- An array is declared and created in almost the same way that objects are declared and created
- BaseType[] ArrayName = new BaseType[size];

- Usually use a for loop to go through an array.

- Array is an object and has an instance variable: length.
- score.length is 5.
- Index is from 0 to length-1, other indexs would cause an out-of-bounds error.

- Initialize with declare: int[] age = {2, 12, 1} (age.length is 3)
- Or initialize with a for loop
- If not initialized, they will automatically be initialized to the default value for their base type

```java
import java.util.Scanner;

public class ArrayOfScores
{
  /**
  Reads in 5 scores and shows how much each
  score differs from the highest score.
  */
  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    double[] score = new double[5];
    int index;
    double max;

    System.out.println("Enter 5 scores:");
    score[0] = keyboard.nextDouble();
    max = score[0];
    for (index = 1; index < 5; index++)
    {
      score[index] = keyboard.nextDouble();
      if (score[index] > max)
        max = score[index];
      //max is the largest of the values score[0], ..., score[index]
    }

    System.out.println("The highest score is " + max);
    System.out.println("The scores are:");
    for (index = 0; index < 5; index++)
      System.out.println(score[index] +
                         " differs from max by " + (max - score[index]));
  }
}
```

Arrays and Reference

- Arrays are objects
- A variable of an array type holds the address of where the array object is stored in memory
- Since an array is a reference type, the behavior of arrays with respect to assignment (=), equality testing (==), and parameter passing are the same as that described for classes
- the assignment operator (=) copies this memory address
- the equality operator (==) tests two arrays share the same address, not values.
- Need to write methods to check length and each values to check if they are equal.
- The heading for the main method of a program has a parameter for an array of String
- You can return an array in the method.

- Date[] holidayList = new Date[20];
- Each of these indexed variables are automatically initialized to null
- for (int i=0; i<holidayList.length; i++) { holidayList[i] = new Date(); }
- Both array indexed variables and entire arrays can be used as arguments to methods

Arrays and Reference

```
public static boolean equalsArray(int[] a, int[] b)
{
  if (a.length != b.length)  return false;
  else
  {
    int i = 0;
    while (i < a.length)
    {
      if (a[i] != b[i])
        return false;
      i++;
    }
  }
  return true;
}
```

```java
public class Ticket {

    public static void myMethod(double a){
        a= a+1;
        }

    public static void main(String[] args) {
        double n = 0.0;
        double[] a = new double[10];//all elements are initialized
        int i = 3;


        myMethod(n);
        myMethod(a[3]);
        myMethod(a[i]);

        System.out.println(n);
        System.out.println(a[3]);
        System.out.println(a[i]);


    }

}
```

```
<terminated> Ticket [.
0.0
0.0
0.0
```

```java
    public static void doubleElements(double[] a) {

    for (int i = 0; i < a.length; i++)
        a[i] = a[i]*2;
    }

    public static void main(String[] args) {
        double[] a = {1.2, 2.2, 3.4};
        doubleElements(a);
        for (int i=0; i<a.length; i++) {
            System.out.println(a[i]);
        }

    }

}
```

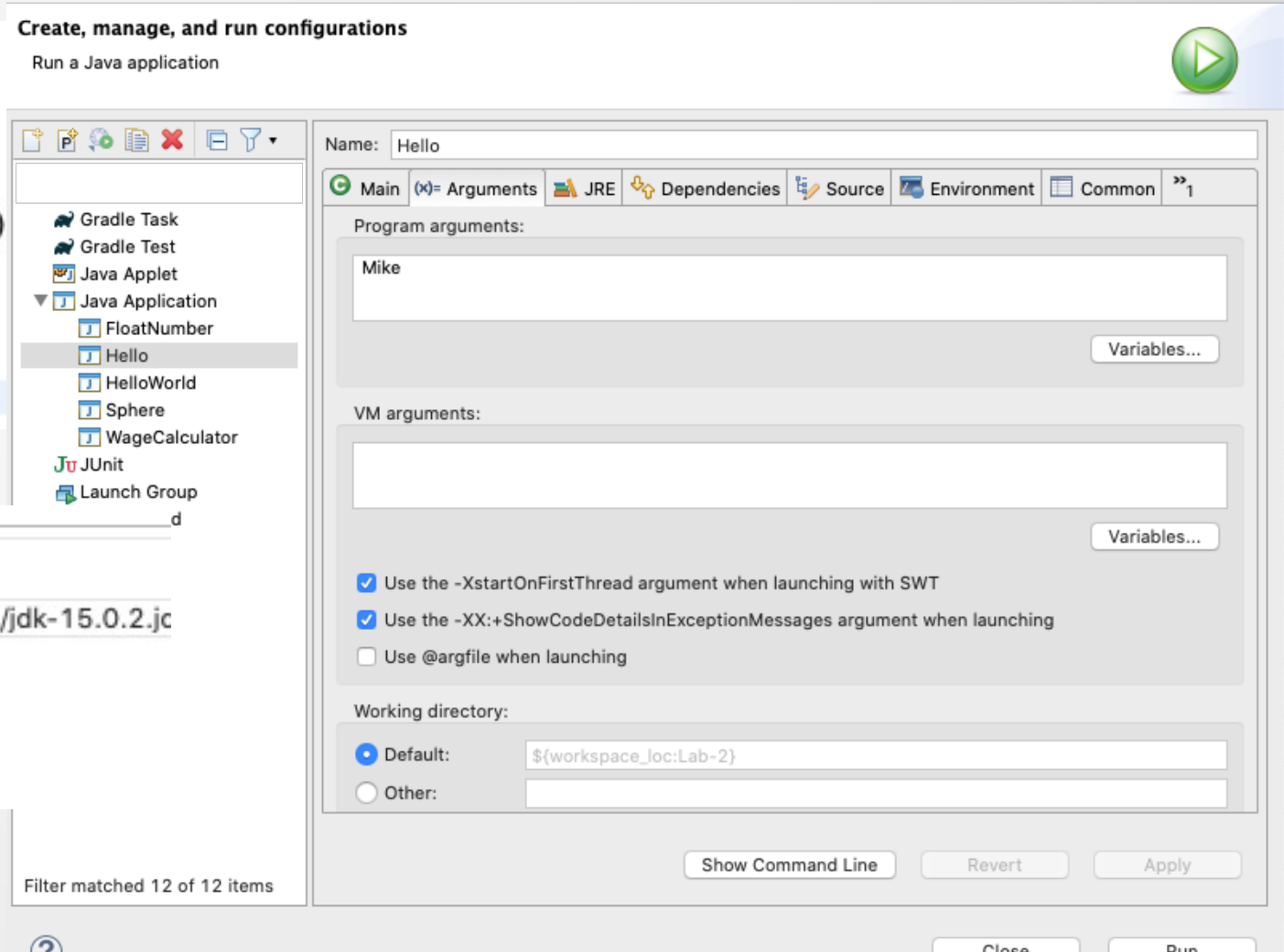
Outline Console

```
<terminated> Ticket [Java Applicat
2.4
4.4
6.8
```

Command Line inputs/arguments

- First argument: args[0], second argument: args[1]
- Each of these arguments is a string (need to convert to other types for use)

```
public class Hello {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        System.out.println("Hello, " + args[0] + "!");

    }

}
```



```java
    public static int[] incrementArray(int[] a, int increment) {
        int[] temp = new int[a.length];
//      int[] temp = a;
        int i;
        for (i = 0; i < a.length; i++) {
            temp[i] = a[i] + increment;
    }
     return temp;
}

    public static void main(String[] args) {
        int[] a = {1, 2, 3};
        int increment = 3;
        int[] b = incrementArray(a, increment);
        for (int i = 0; i < a.length; i++) {
            System.out.println(b[i]);
        }
        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }

    }

}
```

Outline | Console

<terminated> Ticket [Java Application] /Libr

```
4
5
6
1
2
3
```

```java
14
15      public static int[] incrementArray(int[] a, int increment) {
16  //      int[] temp = new int[a.length];
17          int[] temp = a;
18          int i;
19          for (i = 0; i < a.length; i++) {
20              temp[i] = a[i] + increment;
21      }
22       return temp;
23  }
24
25      public static void main(String[] args) {
26          int[] a = {1, 2, 3};
27          int increment = 3;
28          int[] b = incrementArray(a, increment);
29          for (int i = 0; i < a.length; i++) {
30              System.out.println(b[i]);
31          }
32          for (int i = 0; i < a.length; i++) {
33              System.out.println(a[i]);
34          }
35
36      }
37
38  }
39
```

<terminated> Ticket [Java

```
4
5
6
4
5
6
```

for-each loop to go through array

```
public static void main(String[] args) {
    int[] a = {1, 2, 3};

    for (int i = 0; i < a.length; i++) {
        a[i]++;
    }
/   for (int element: a) {
/       element = element+1;
/   }


    for (int i = 0; i < a.length; i++) {
        System.out.println(a[i]);
    }

    for (int element: a) {
        System.out.println(element);
    }

}
```

```
<terminated> Ticket
2
3
4
2
3
4
```

```
public static void main(String[] args) {
    int[] a = {1, 2, 3};

//      for (int i = 0; i < a.length; i++) {
//          a[i]++;
//      }
        for (int element: a) {
            element = element+1;
        }


        for (int i = 0; i < a.length; i++) {
            System.out.println(a[i]);
        }

        for (int element: a) {
            System.out.println(element);
        }

    }

}
```

```
<terminated> Ticket [Java
1
2
3
1
2
3
```

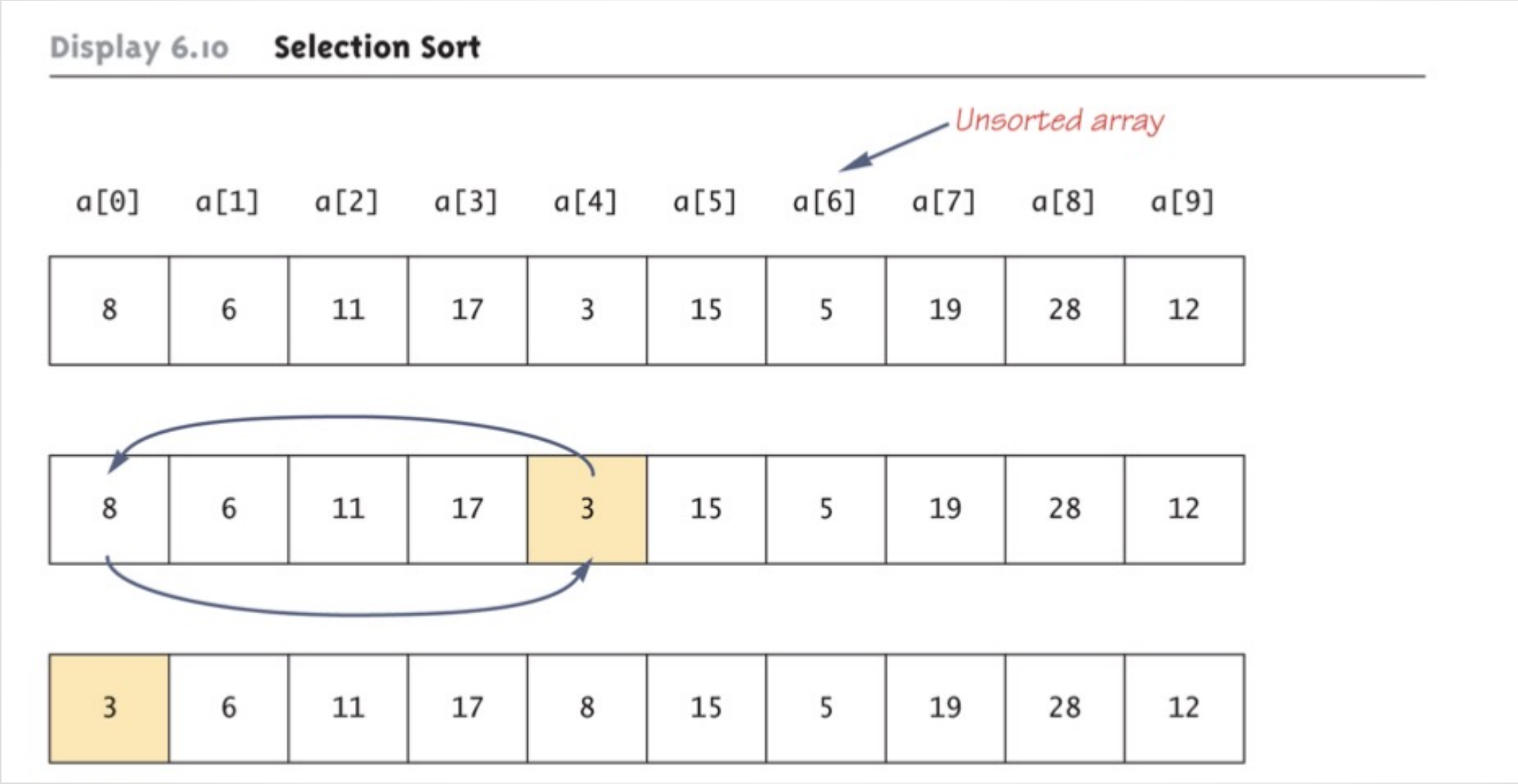
Avoid privacy leak for array

- Array is a mutable object, don't give an array directly, but create a new one and copy all values.

```
public double[] getArray()
{
  double[] temp = new double[count];
  for (int i = 0; i < count; i++)
    temp[i] = a[i];
  return temp
}
```

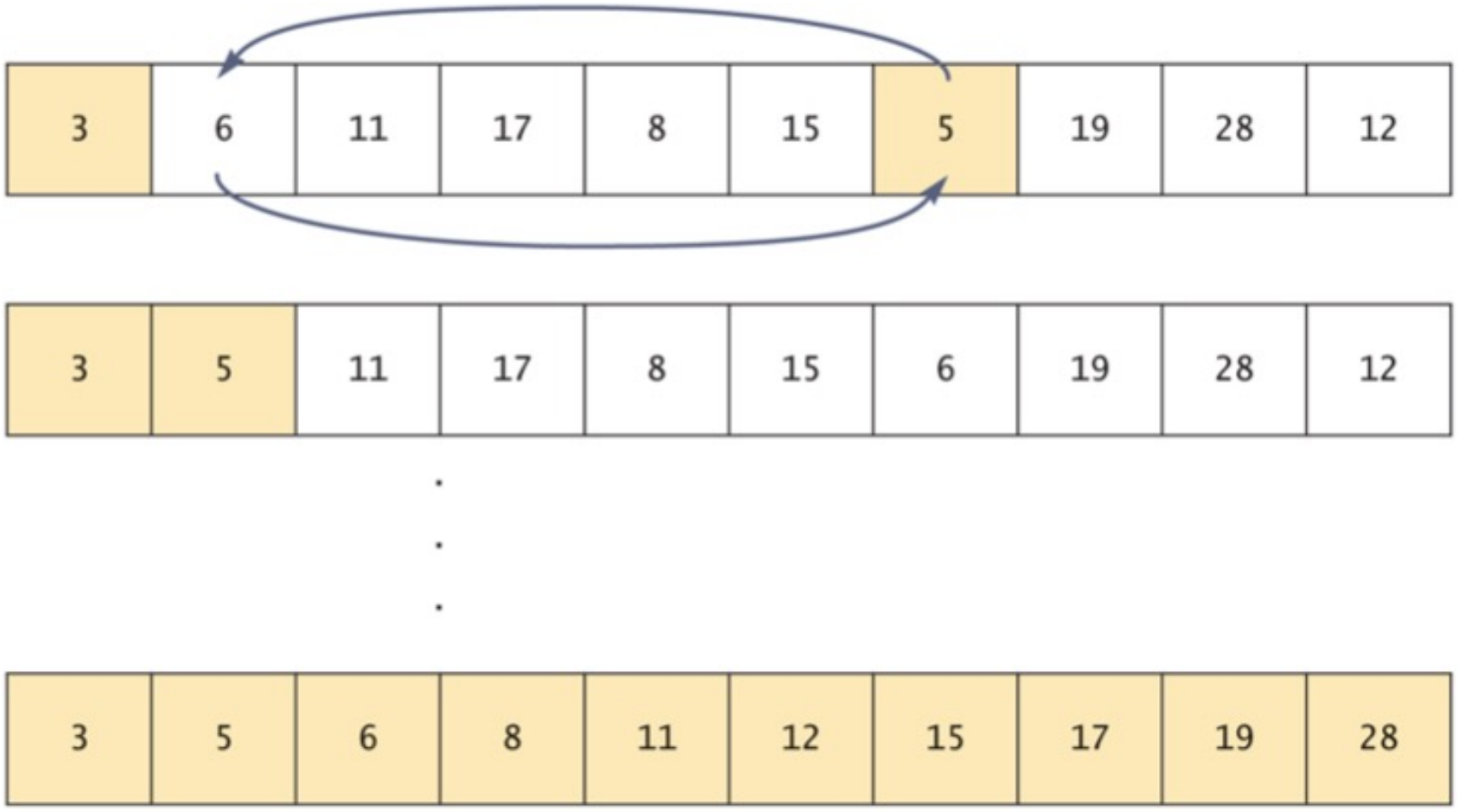
```
public ClassType[] getArray()
{
  ClassType[] temp = new ClassType[count];
  for (int i = 0; i < count; i++)
    temp[i] = new ClassType(someArray[i]);
  return temp;
}
```

Selection Sort in Array



Selection Sort in Array

Display 6.10 **Selection Sort**



Selection Sort in Array

```
24
25      public static void selectionSort(int[] array){
26          int temp;
27          int index;
28          for (int i=0; i < array.length; i++){
29              temp = array[i];
30              index = i;
31              for ( int j = i + 1; j < array.length ; j ++){
32                  if (temp > array[j]){
33                      temp = array[j];
34                      index = j;
35                  }
36              }
37
38              array[index] = array[i];
39              array[i] = temp;
40          }
41      }
42
43      public static void main(String[] args) {
44          int[] a = {8, 6, 11, 17, 3, 15, 5, 19, 28, 12};
45
46          selectionSort(a);
47
48          for (int element: a) {
49              System.out.println(element);
50          }
51
52      }
53
54  }
55
```

```
<terminated> Ticket [Java Application] /
3
5
6
8
11
12
15
17
19
28
```

Enumerated Type

- Work likes String but one variable can only be set to within the fixed items. Perfectly fit with switch

```
public enum WorkDay {
    Monday, TUESDAY, WESDESDAY, THURSDAY, FRIDAY, monday

}
```

```
    public static void main(String[] args) {
        WorkDay startDay = WorkDay.Monday;
        WorkDay endDay = WorkDay.FRIDAY;

        System.out.println("Work starts on " + startDay);
        System.out.println("Work starts on " + WorkDay.monday);
        System.out.println("Work ends on " + endDay);
        WorkDay day1 = WorkDay.valueOf("monday");
        System.out.println("One day is " + day1);
//      System.out.println("One day is " + day2);

    }

}
```

ArrayList

- An object that can works like array, but change length and less effeciently.
- The class ArrayList is implemented using an array as a private instance variable – When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array
- An ArrayList is less efficient than an array
  It does not have the convenient square bracket notation
  The base type of an ArrayList must be a class type or interface type (or other reference type): it cannot be a primitive type
- ArrayList come with many powerful methods you can use.

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

        ArrayList<Double> list = new ArrayList<Double>();
        list.add(0.1);
        list.add(0,0.2);
        list.add(0,0.3);
        list.add(0,0.4);
        list.add(0,0.5);

        for(double i : list) {
            System.out.println(i);
        }
        System.out.println("----------");

        Collections.shuffle(list);

        for(double i : list) {
            System.out.println(i);
        }


//      System.out.println(list.get(0));
```

```
<terminated> Ticket [Java Ap
0.5
0.4
0.3
0.2
0.1
----------
0.2
0.4
0.5
0.1
0.3
```

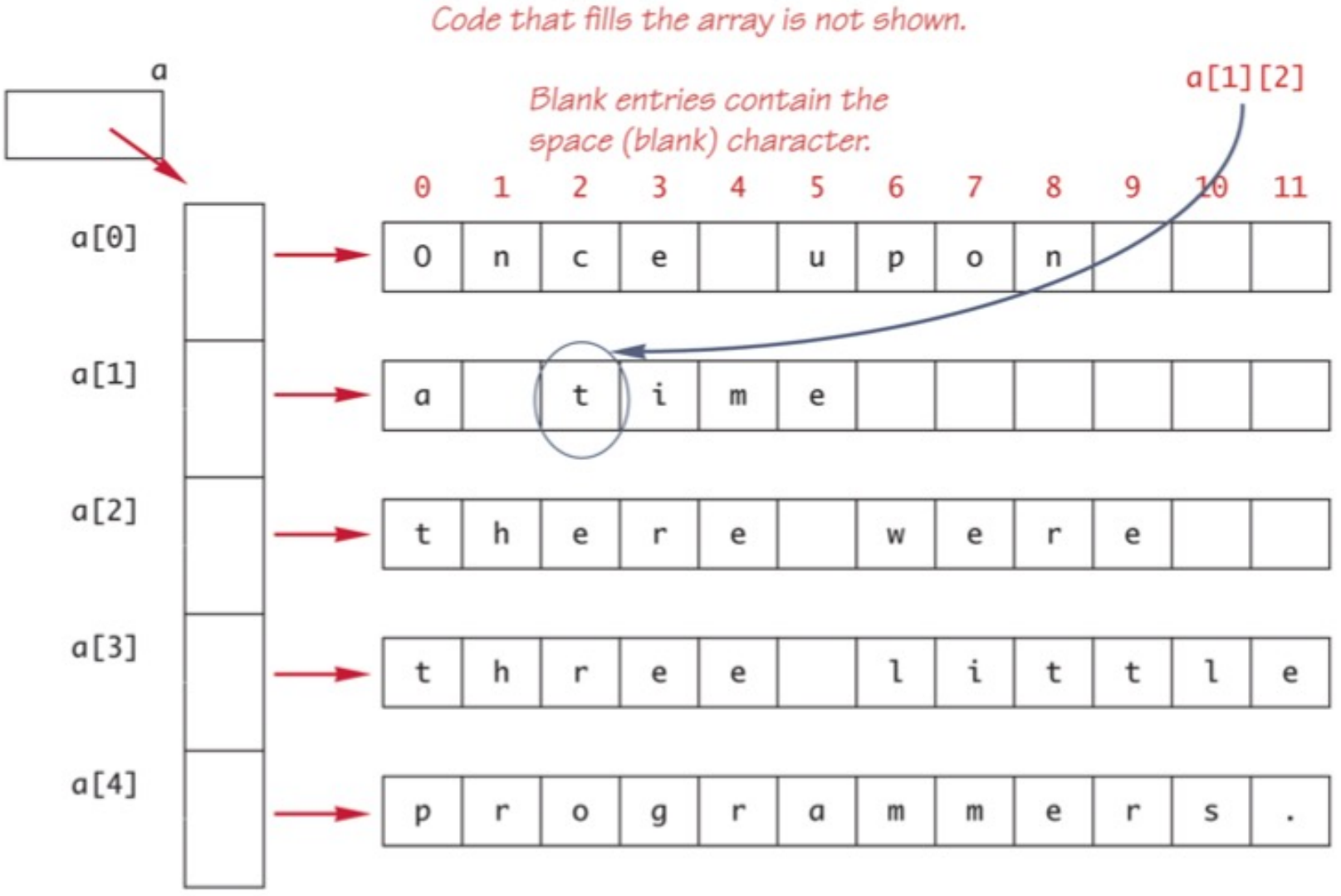
Multidimendional Arrays

- Multidimensional arrays are declared and created in the same ways as one-dimensional arrays.
- double[][]table = new double[100][10];
- int[][][] figure = new int[10][20][30];
- Person[][] = new Person[10][100];

- Two-dimensional array is array of arrays
- char[][] page = new char[30][100];
- page.length is equal to 30
- page[0].length is equal to 100

- Ragged array: an array that has a
- different number of elements per row

```
char[][] a = new char[5][12];
```



```
int row, column;
for (row = 0; row < 5; row++)
{
    for (column = 0; column < 12; column++)
       System.out.print(a[row][column]);
    System.out.println();
}
```

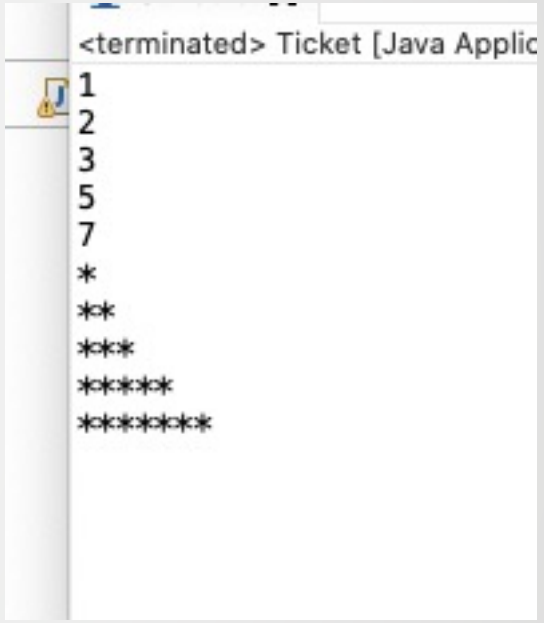
We will see that these can and should be replaced with expressions involving the `length` *instance variable.*

Produces the following output:

```
Once upon
a time
there were
three little
programmers.
```

Multidimendional Arrays

```
5
6      public static void main(String[] args) {
7          int[][] numbers = new int[5][];
8          numbers[0] = new int[1];
9          numbers[1] = new int[2];
0          numbers[2] = new int[3];
1          numbers[3] = new int[5];
2          numbers[4] = new int[7];
3          System.out.println(numbers[0].length);
4          System.out.println(numbers[1].length);
5          System.out.println(numbers[2].length);
6          System.out.println(numbers[3].length);
7          System.out.println(numbers[4].length);
8
9          for (int row=0; row<numbers.length;row++) {
0              for(int col=0; col<numbers[row].length; col++) {
1                  System.out.print('*');
2
3              }
4              System.out.print('\n');
5          }
6
7      }
8
9  }
```



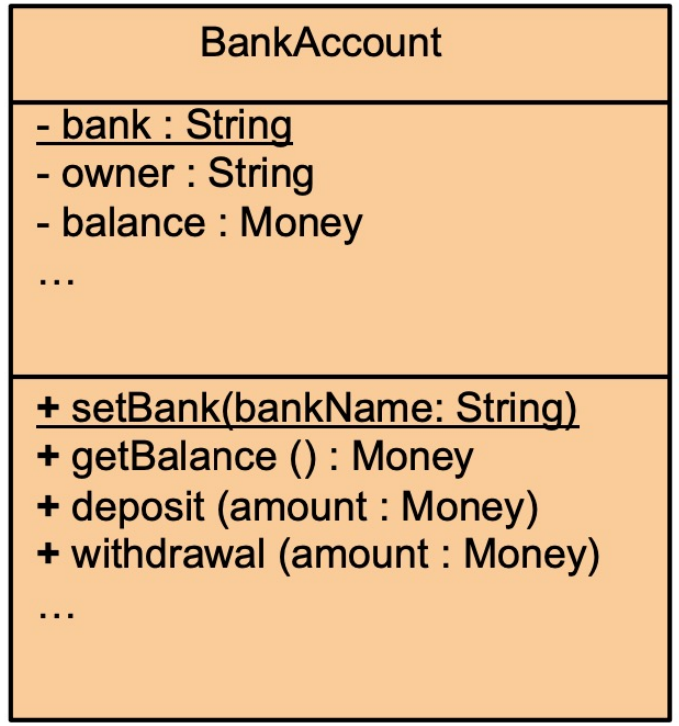
Unified Modeling Language(UML)

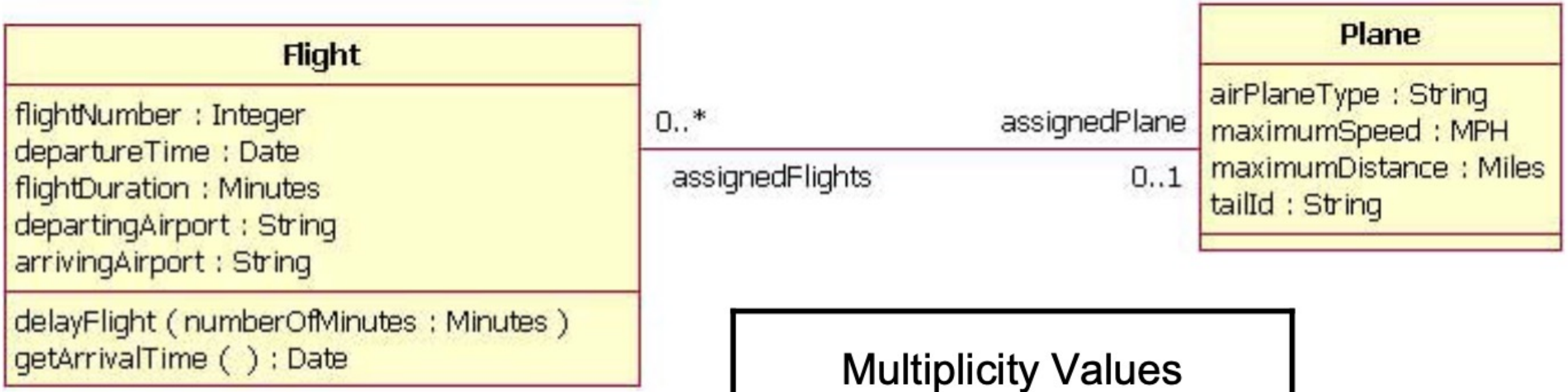
- A type of structure diagram
- Describe the structure of a system by showing the system's classes, attributes, methods and relationship.

- The data specification for each piece of data in a UML diagram consists of
- its name, followed by a colon, followed by its type

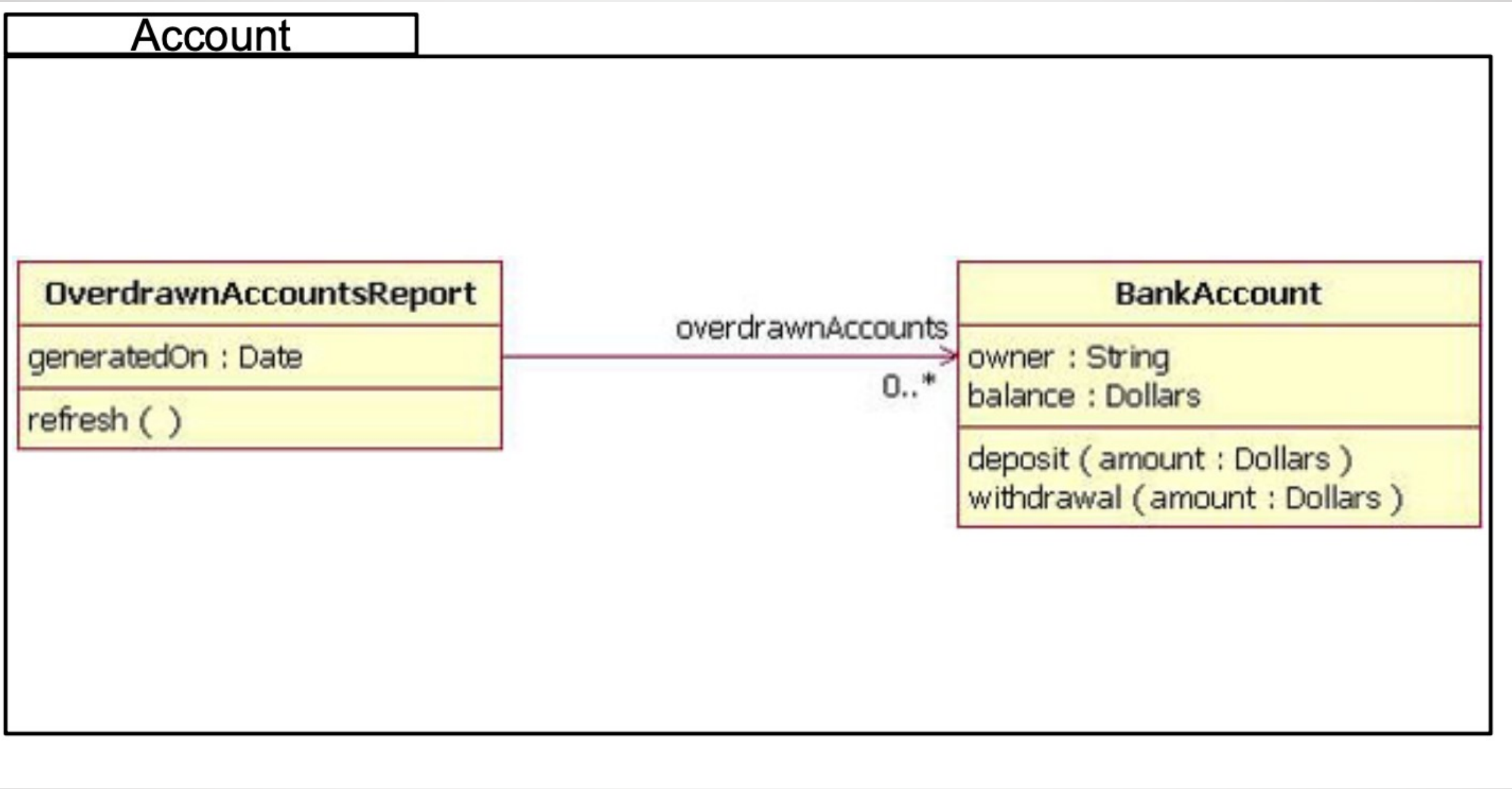
- Each name is preceded by a character that specifies its access type:
- A minus sign (-) indicates private access
- A plus sign (+) indicates public access
- A sharp (#) indicates protected access
- A tilde (~) indicates package access

- Missing members are indicated with an ellipsis (three dots)
- UML has various ways to indicate the information flow from
- one class object to another using different sorts of annotated arrows
- UML has annotations for class groupings into packages,
- for inheritance, and for other interactions





Multiplicity Values	
Indicator	Meaning
n	exactly n
*	zero or many
0..n	zero to n
m..n	m to n


Account
OverdrawnAccountsReport
generatedOn : Date
refresh ( )
overdrawnAccounts
0..*
BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)

package

- Java uses packages to form libraries of classes
- A package is a group of classes that have been placed in a directory or folder, and that can be used in any program that includes an import statement that names the package
- The import statement must be located at the beginning of the program file
- import java.util.Scanner; to use Scanner object

- import java.util.* makes all the classes in a package available instead of just one class
- Drawbacks of using *
- Worse readability of code due to lack of info of which package is used
- Possibly longer compilation time
- Larger possibility of conflict of package names with other
- To make a package, group all the classes together into a single directory (folder), and add the following package statement to the beginning of each java file for those classes
- package package_name;
- Only the .class files must be in the directory or folder, the .java files are optional
- Only blank lines and comments may precede the package statement
- If there are both import and package statements, the package statement must precede any import statements

CLASSPATH

- Java needs two things to find the directory for a package: the name of the package and the value of the CLASSPATH variable
- When a package is stored in a subdirectory of the directory containing another package, importing the enclosing package does not import the subdirectory package
- import utilities.numericstuff.*;
- import utilities.numericstuff.statistical.*;
- All the classes in the current directory belong to an unnamed package called the default package
- As long as the current directory (.) is part of the CLASSPATH variable, all the classes in the default package are automatically available to a program
- If the CLASSPATH variable is set, the current directory must be included as one of the alternatives
- If the CLASSPATH variable is not set, then all the class files for a program must be put in the current directory
- The class path can be manually specified when a class is compiled
- add –classpath followed by the desired class path will compile the class, overriding any previous CLASSPATH setting
- use the –classpath option again when the class is run
- packages provide a way to deal with name clashes: a situation in which two classes have the same name
- This ambiguity can be resolved by using the fully qualified name (i.e., precede the class name by its package name) to distinguish between each class
- package_name.ClassName

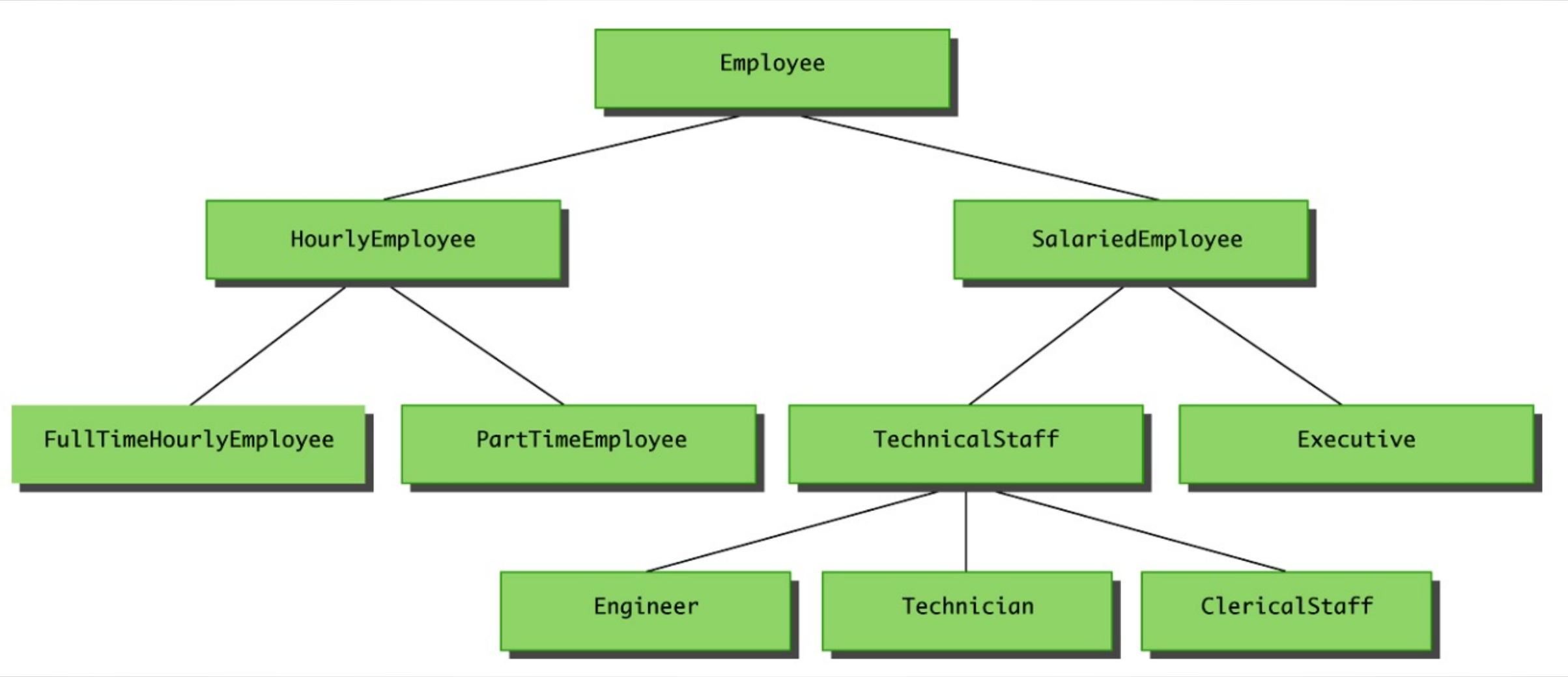
javadoc

- Java has a program called javadoc that automatically extracts the interface from a class definition and produces documentation
- The javadoc program extracts class headings, the headings for some comments, and headings for all public methods, instance variables, and static variables
- In the normal default mode, no method bodies or private items are extracted
- To extract a comment, the following must be true:
- The comment must immediately precede a public class or method definition, or some other public item
- The comment must be a block comment, and the opening /* must contain an extra*(/** . . . */)
- In addition to any general information, the comment preceding a public method definition should include descriptions of parameters, any value returned, and any exceptions that might be thrown

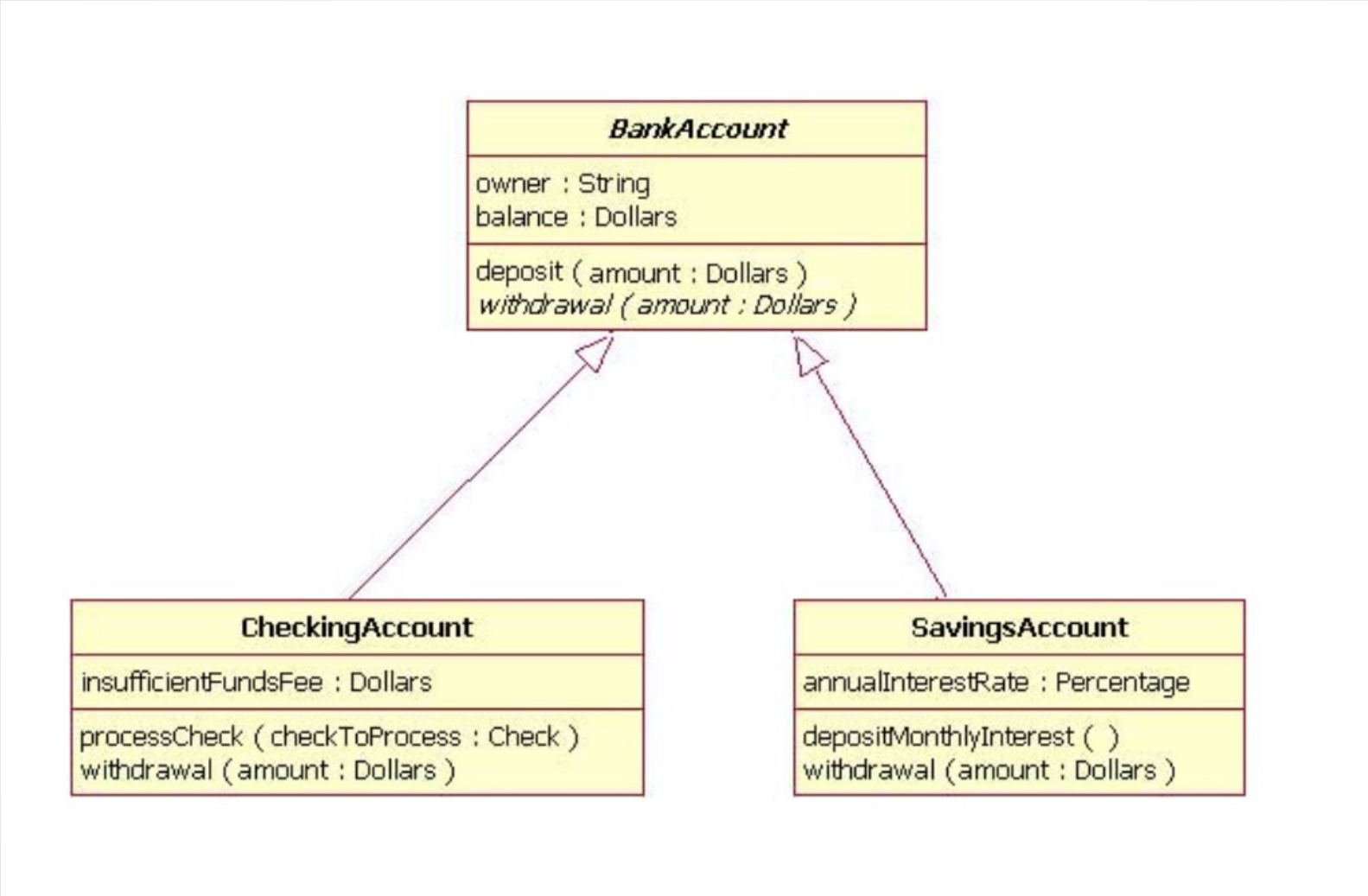
Inheritance

- Inheritance is one of the main techniques of object-oriented programming (OOP)
- a very general form of a class is first defined and compiled, and then more specialized versions of the class are defined by adding instance variables and methods
- The specialized classes are said to inherit the methods and instance variables of the general class
- Inheritance is the process by which a new class is created from another class
- The new class is called a derived/child/sub class
- The original class is called the base/parent/super class
- A derived class automatically has all the instance variables and methods that the base class has, and it can have additional methods and/or instance variables as well
- public class HourlyEmployee extends Employee
- Class Employee defines the instance variables name and hireDate in its class definition
- Class HourlyEmployee has additional instance variables wageRate and hours that are specified in its class definition
- Inherited Members: The derived class inherits all the public methods, all the instance variables, and all the static variables from the base class
- a class that is a parent of a parent . . . of another class is called an ancestor class
- If class A is an ancestor of class B, then class B can be called a descendent of class A

Inheritance



Inheritance



Override

- a derived class inherits methods from the base class, it can change or override an inherited method if necessary
- the type returned may not be changed when overriding a method
- if it is a class type, then the returned type may be changed to that of any descendent class of the returned type
- public class BaseClass
- {...
- public Employee getSomeone(int someKey) ...

- public class DerivedClass extends BaseClass
- {...
- public HourlyEmployee getSomeone(int someKey) ...
- The access permission of an overridden method can be changed from private in the base class to public (or some other more permissive access) in the derived class
- the access permission of an overridden method can not be changed from public in the base class to a more restricted access permission in the derived class
- When a method is overridden, the new method definition given in the derived class has the exact same number and types of parameters as in the base class
- When a method in a derived class has a different signature from the method in the base class, that is overloading

final

- If the modifier final is placed before the definition of a method, then that method may not be redefined in a derived class
- It the modifier final is placed before the definition of a class, then that class may not be used as a base class to derive other classes
- A derived class uses a constructor from the base class to initialize all the data inherited from the base class
- public derivedClass(int p1, int p2, double p3) {
- super(p1, p2);
- instanceVariable = p3;
- }
- A call to the base class constructor can never use the name of the base class, but uses the keyword super instead
- A call to super must always be the first action taken in a constructor definition
- An instance variable cannot be used as an argument to super
- If a derived class constructor does not include an invocation of super, then the no-argument constructor of the base class will automatically be invoked
- Within the definition of a constructor for a class, this can be used as a name for invoking another constructor in the same class
- If it is necessary to include a call to both super and this, the call using this must be made first, and then the constructor that is called must call super as its first action
- a no-argument constructor uses this to invoke an explicit-value constructor
- public ClassName()
- {
- this(argument1, argument2);
- }

- public ClassName(type1 param1, type2 param2)
- { ...}

this

- public HourlyEmployee()
- { this("No name", new Date(), 0, 0); }
- public HourlyEmployee(String theName, Date theDate, double theWageRate, double theHours)
- an object of a derived class has the type of every one of its ancestor classes
- an object of a derived class can be assigned to a variable of any ancestor type
- a derived class object can be used anyplace that an object of any of its ancestor types can be used
- An ancestor type can never be used in place of one of its derived types
- An instance variable that is private in a base class is not accessible by name in the definition of a method in any other class, not even in a method definition of a derived class
- If private instance variables of a class were accessible in method definitions of a derived class, then anytime someone wanted to access a private instance variable, they would only need to create a derived class, and access it in a method of that class
- The private methods of the base class are like private variables in terms of not being directly available
- This should not be a problem because private methods should just be used as helping methods
- Thanks to inheritance, most of the standard Java library classes can be enhanced by defining a derived class with additional methods

Display 7.7 EnhancedStringTokenizer

```
1  import java.util.StringTokenizer;
2
3  public class EnhancedStringTokenizer extends StringTokenizer
4  {
5      private String[] a;
6      private int count;

7      public EnhancedStringTokenizer(String theString)
8      {
9          super(theString);
10         a = new String[countTokens()];
11         count = 0;
12     }

13     public EnhancedStringTokenizer(String theString, String delimiters)
14     {
15         super(theString, delimiters);
16         a = new String[countTokens()];
17         count = 0;
18     }
```

The method **countTokens** *is inherited and is not overridden.*

(continued)

Display 7.7 EnhancedStringTokenizer

```
19      /**
20       Returns the same value as the same method in the StringTokenizer class,
21       but it also stores data for the method tokensSoFar to use.
22      */
23      public String nextToken()
24      {
25          String token = super.nextToken();
26          a[count] = token;
27          count++;
28          return token;
29      }
```

This method **nextToken** *has its definition overridden.*

super.nextTokens *is the version of* **nextToken** *defined in the base class* **StringTokenizer**. *This is explained more fully in Section 7.3.*

(continued)

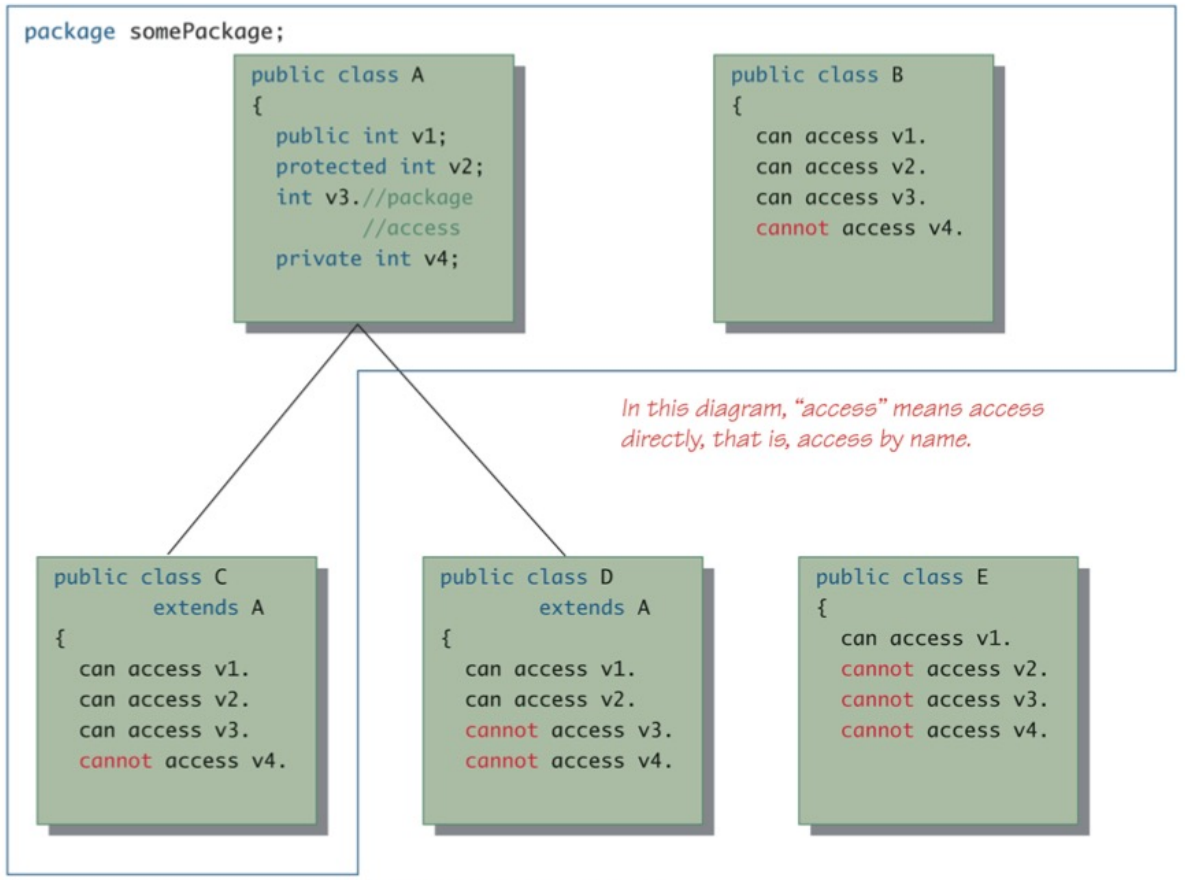
Display 7.7 EnhancedStringTokenizer

```
42      /**
43       Returns an array of all tokens produced so far.
44       Array returned has length equal to the number of tokens produced so far.
45      */
46      public String[] tokensSoFar()
47      {
48          String[] arrayToReturn = new String[count];
49          for (int i = 0; i < count; i++)
50              arrayToReturn[i] = a[i];
51          return arrayToReturn;
52      }
53  }
```

tokensSoFar *is a new method.*

Access Modifiers

Display 7.9 **Access Modifiers**



protected

- If a method or instance variable is modified by protected (rather than public or private), then it can be accessed by name Inside any class derived from it
- An instance variable or method definition that is not preceded with a modifier has package access
- Instance variables or methods having package access can be accessed by name inside the definition of any class in the same package
- It is only valid to use super to invoke a method from a direct parent
- For example, if the Employee class were derived from the class Person, and the HourlyEmployee class were derived form the class Employee, it would not be possible to invoke the toString method of the Person class within a method of the HourlyEmployee class super.super.toString() // ILLEGAL!
- every class is a descendent of the class Object
- The class Object is in the package java.lang which is always imported automatically
- some library methods accept an argument of type Object so they can be used with an argument that is an object of any class
- The class Object has some methods that every Java class inherits
- For example,the equals and toString methods
- Every object inherits the same getClass() method from the Object class
- This method is marked final, so it cannot be overridden
- An invocation of getClass() on an object returns a representation only of the class that was used with new to create the object
- The results of any two such invocations can be compared with == or != to determine whether or not they represent the exact same class
- The instanceof operator checks if an object is of the type given as its second argument
- it will return true if Object is the type of any descendent class of ClassName

Polymorphism

- There are three main programming mechanisms that constitute object-oriented programming (OOP)
- Encapsulation
- Inheritance
- Polymorphism
- Polymorphism is the ability to associate many meanings to one method name
- It does this through a special mechanism known as late binding or dynamic binding
- Inheritance allows a base class to be defined, and other classes derived from it
- Polymorphism allows changes to be made to method definitions in the derived classes, and have those changes apply to the software written for the base class
- The process of associating a method definition with a method invocation is called binding
- If the method definition is associated with its invocation when the code is compiled, that is called early binding or static binding
- If the method definition is associated with its invocation when the method is invoked (at run time), that is called late binding or dynamic binding
- Java uses late binding for all methods (except private, final, and static methods)
- Because of late binding, a method can be written in a base class to perform a task, even if portions of that task aren't yet defined

Late Binding

- The **`Sale`** class **`lessThan`** method
  - Note the **`bill()`** method invocations:

```
public boolean lessThan (Sale otherSale)
{
  if (otherSale == null)
  {
    System.out.println("Error: null object");
    System.exit(0);
  }
  return (bill( ) < otherSale.bill( ));
}
```

Late Binding

- The **Sale** class **bill()** method:

```
public double bill( )
{
  return price;
}
```

- The **DiscountSale** class **bill()** method:

```
public double bill( )
{
      double discountedPrice = getPrice() * (1-discount/100);
      return discountedPrice + discountedPrice * SALES_TAX/100;
}
```

Late Binding

- Given the following in a program:

```
. . .
Sale simple = new sale("floor mat", 10.00);
DiscountSale discount = new
        DiscountSale("floor mat", 11.00, 10);
. . .
if (discount.lessThan(simple))
  System.out.println("$" + discount.bill() +
            " < " + "$" + simple.bill() +
            " because late-binding works!");
. . .
```

  - Output would be:

$9.90 < $10 because late-binding works!

Upcasting

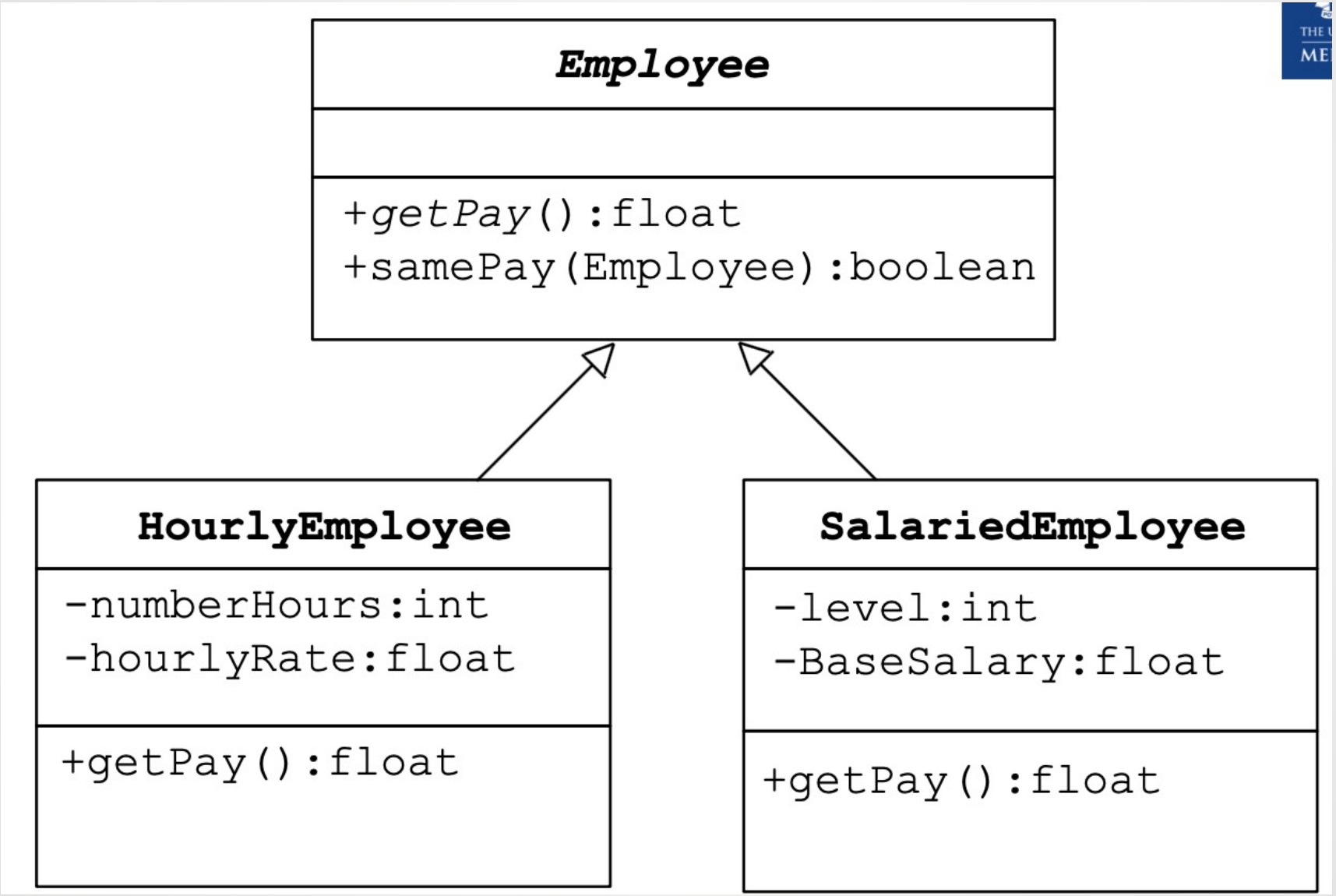
- Upcasting is when an object of a derived class is assigned to a variable of a base class (or any ancestor class)
- Downcasting is when a type cast is performed from a base class to a derived class (or from any ancestor class to any descendent class)

Abstract Class

- In order to postpone the definition of a method, Java allows an abstract method to be declared
- An abstract method has a heading, but no method body
- The body of the method is defined in the derived classes
- The class that contains an abstract method is called an abstract class An abstract class must have the modifier abstract included in its class heading
- public abstract double getPay();
- public abstract void doIt(int count);
- If a derived class of an abstract class adds to or does not define all of the abstract methods, then it is abstract also, and must add abstract to its modifier
- A class that has no abstract methods is called a concrete class

- Abstract Class and Interface
- When we talk about abstract classes we are defining characteristics of an object type; specifying what an object is.
- When we talk about an interface and define capabilities that we promise to provide, we are talking about establishing a contract about what the object can do.

Abstract Class



Interface

- An interface specifies a set of methods that any class that implements the interface must have
- An interface serves a function similar to a base class, though it is not a base class
- Some languages allow one class to be derived from two or more different base classes
- multiple inheritance is not allowed in Java
- Java's way of approximating multiple inheritance is through interfaces
- An interface and all of its method headings should be declared public
- Abstract classes may implement one or more interfaces
- Any method headings given in the interface that are not given definitions are made into abstract methods
- A concrete class must give definitions for all the method headings given in the abstract class and the interface

Display 13.1 The Ordered Interface

```
1  public interface Ordered
2  {
3      public boolean precedes(Object other);

4      /**
5       For objects of the class o1 and o2,
6       o1.follows(o2) == o2.preceded(o1).
7      */
8      public boolean follows(Object other);
9  }
```

Do not forget th
the end of the m

Neither the compiler nor the run-time system will do a
satisfied. It is only advisory to the programmer implen

```
1  public class OrderedHourlyEmployee
2          extends HourlyEmployee implements Ordered
3  {
4      public boolean precedes(Object other)
5      {
6          if (other == null)
7              return false;
8          else if (!(other instanceof OrderedHourlyEmployee))
9              return false;
10         else
11         {
12             OrderedHourlyEmployee otherOrderedHourlyEmployee =
13                                 (OrderedHourlyEmployee)other;
14             return (getPay() < otherOrderedHourlyEmployee.getPay());
15         }
16     }

       public boolean follows(Object other)
       {
           if (other == null)
               return false;
           else if (!(other instanceof OrderedHourlyEmployee))
               return false;
           else
           {
               OrderedHourlyEmployee otherOrderedHourlyEmployee =
                                   (OrderedHourlyEmployee)other;
               return (otherOrderedHourlyEmployee.precedes(this));
           }
       }
   }
```

Although `getClass` works better than `instanceof` for defining `equals`, `instanceof` works better in this case. However, either will do for the points being made here.

Display 13.3 An Abstract Class Implementing an Interface

```
1  public abstract class MyAbstractClass implements Ordered
2  {
3      int number;
4      char grade;
5
6      public boolean precedes(Object other)
7      {
8          if (other == null)
9              return false;
10         else if (!(other instanceof HourlyEmployee))
11             return false;
12         else
13         {
14             MyAbstractClass otherOfMyAbstractClass =
15                                          (MyAbstractClass)other;
16             return (this.number < otherOfMyAbstractClass.number);
17         }
18     }

19     public abstract boolean follows(Object other);

20 }
```

Interface

- Like classes, an interface may be derived from a base interface
- The derived interface must include the phrase extends BaseInterfaceName
- A concrete class that implements a derived interface must have definitions for any methods in the derived interface as well as any methods in the base interface
- An interface can contain defined constants in addition to or instead of method headings
- Any variables defined in an interface must be public, static, and final
- a class can have only one base class
- a class may implement any number of interfaces
- If a class definition implements two inconsistent interfaces, then that is an error, and the class definition is illegal
- One type of inconsistency will occur if the interfaces have constants with the same name, but with different values
- Another type of inconsistency will occur if the interfaces contain methods with the same name but different return types

Comparable Interface

- The Comparable interface is in the java.lang package, and so is automatically available to any program
- It has only the following method heading that must be implemented: public int compareTo(Object other);
- The method compareTo must return
- A negative number if the calling object "comes before" the parameter other
- A zero if the calling object "equals" the parameter other
- A positive number if the calling object "comes after" the parameter other
- Both the Double and String classes implement
- the Comparable interface
- Interfaces apply to classes only
- A primitive type (e.g. double) cannot implement an interface

Exception Handling

- Java provides a mechanism that signals when something unusual happens: throwing an exception
- provide code that deals with the exceptional case: handling the exception
- A throw statement is similar to a method call: throw new ExceptionClassName(SomeString);
- Instead of calling a method, a throw statement calls a catch block

```
. . . // method code
try
{
  . . .
  throw new Exception(StringArgument);
  . . .
}
catch(Exception e)
{
  String message = e.getMessage();
  System.out.println(message);
  System.exit(0);
} . . .
```

Exception Handling

- When an exception is thrown, the catch block begins execution
- The catch block has only one parameter
- The exception object thrown is plugged in for the catch block parameter
- e is called the catch block parameter
- It specifies the type of thrown exception object that the catch block can catch
- It provides a name (for the thrown object that is caught) on which it can operate in the catch block
- Numerous predefined exception classes are included in the standard packages that come with Java
- IOException
- NoSuchMethodException
- FileNotFoundException
- Instead of using a predefined class, exception classes can be programmer-defined
- Every exception class to be defined must be a derived class of some already defined exception class

Exception Handling

```
1   public class DivisionByZeroException extends Exception
2   {
3       public DivisionByZeroException()
4       {
5           super("Division by Zero!");
6       }

7       public DivisionByZeroException(String message)
8       {
9           super(message);
10      }
11  }
```

You can do more in an exception constructor, but this form is common.

super is an invocation of the constructor for the base class Exception.

Exception Catch

- When catching multiple exceptions, the order of the catch blocks is important
- When an exception is thrown in a try block, the catch blocks are examined in order
- The first one that matches the type of the exception thrown is the one that is executed
- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
- If a method can throw an exception but does not catch it, it must provide a warning in the heading
- The process of including an exception class in a throws clause is called declaring the exception
- If a method throws an exception and does not catch it, then the method invocation ends immediately
- If a method can throw more than one type of exception, then separate the exception types by commas
- public void aMethod() throws AnException, AnotherException

```
catch (Exception e)
{ . . . }
catch (NegativeNumberException e)
{ . . . }
```

Exception Catch

```
public void someMethod()
            throws SomeException
{
  . . .
  throw new
        SomeException(SomeArgument);
  . . .
}
```

```
public void otherMethod()
{
  try
  {
    someMethod();
    . . .
  }
  catch (SomeException e)
  {
    CodeToHandleException
  }
  . . .
}
```

Exception Catch

Display 9.11 An Exception Controlled Loop

```
1   import java.util.Scanner;
2   import java.util.InputMismatchException;

3   public class InputMismatchExceptionDemo
4   {
5       public static void main(String[] args)
6       {
7           Scanner keyboard = new Scanner(System.in);
8           int number = 0; //to keep compiler happy
9           boolean done = false;
```

Exception Catch

Display 9.11 An Exception Controlled Loop

```
10          while (! done)
11          {
12              try
13              {
14                  System.out.println("Enter a whole number:");
15                  number = keyboard.nextInt();
16                  done = true;
17              }
18              catch(InputMismatchException e)
19              {
20                  keyboard.nextLine();
21                  System.out.println("Not a correctly written whole number.");
22                  System.out.println("Try again.");
23              }
24          }

25          System.out.println("You entered " + number);
26      }
27  }
```

If `nextInt` *throws an exception, the* `try` *block ends and so the* `boolean` *variable* `done` *is not set to* `true`.

(continued)

File I/O

- A stream is an object that enables the flow of data between a program and some I/O device or file
- Input streams can flow from the keyboard or from a file
- System.in is an input stream that connects to the keyboard
- Output streams can flow to a screen or to a file
- System.out is an output stream that connects to the screen
- Files that are designed to be read by human beings, and that can be read or written with an editor are called text files
- Files that are designed to be read by programs and that consist of a sequence of binary digits are called binary files
- An advantage of text files is that they are usually the same on all computers, so that they can move from one computer to another
- An advantage of binary files is that they are more efficient to process than text files
- The class PrintWriter is a stream class that can be used to write to a text file
- A stream of the class PrintWriter is created and connected to a text file for writing as follows:
- PrintWriter outputStreamName;
- outputStreamName = new PrintWriter(new FileOutputStream(FileName));
- This produces an object of the class PrintWriter that is connected to the file FileName
- The process of connecting a stream to a file is called opening the file
- If the file already exists, then doing this causes the old contents to be lost
- If the file does not exist, then a new, empty file named FileName is created
- After doing this, the methods print and println can be used to write to the file

File I/O

- When a program is finished writing to a file, it should always close the stream connected to that file
- outputStreamName.close();
- This allows the system to release any resources used to connect the stream to the file
- If the program does not close the file before the program ends, Java will close it automatically, but it is safest to close it explicitly
- Output streams connected to files are usually buffered
- Rather than physically writing to the file as soon as possible, the data is saved in a temporary location (buffer)
- When enough data accumulates, or when the method flush is invoked, the buffered data is written to the file all at once
- This is more efficient, since physical writes to a file can be slow
- The method close invokes the method flush, thus insuring that all the data is written to the file
- If a program relies on Java to close the file, and the program terminates abnormally, then any output that was buffered may not get written to the file
- The sooner a file is closed after writing to it, the less likely it is that there will be a problem

IOException

- When performing file I/O there are many situations in which an exception, such as FileNotFoundException, may be thrown
- Many of these exception classes are subclasses of the class IOException
- These exception classes are all checked exception.
- Therefore, they must be caught or declared in a throws clause
- In contrast, the exception classes NoSuchElementException, InputMismatchException, and IllegalStateException are all unchecked exceptions
- Unchecked exceptions are not required to be caught or declared in a throws clause
- To create a PrintWriter object and connect it to a text file for appending, a second argument, set to true, must be used in the constructor for the FileOutputStream object
- outputStreamName = new PrintWriter(new FileOutputStream(FileName, true));
- After this statement, the methods print, println and/or printf can be used to write to the file
- The new text will be written after the old text in the file

Read File

- The class Scanner can be used for reading from the keyboard as well as reading from a text file
- Simply replace the argument System.in(to the Scanner constructor) with a suitable stream that is connected to the text file
- Scanner StreamObject = new Scanner(new FileInputStream(FileName));
- Methods of the Scanner class for reading input behave the same whether reading from the keyboard or reading from a text file
- For example, the nextInt and nextLine methods
- A program that tries to read beyond the end of a file using methods of the Scanner class will cause an exception to be thrown
- instead of having to rely on an exception to signal the end of a file, the Scanner class provides methods such as hasNextInt and hasNextLine

BufferedReader

- The class BufferedReader is a stream class that can be used to read from a text file
- An object of the class Buffered Reader has the methods read and readLine
- A stream of the class BufferedReader is created and connected to a text file as follows:
- BufferedReader readerObject;
- readerObject = new BufferedReader(new FileReader(FileName));
- The readLine method is the same method used to read from the keyboard, but in this case it would read from a file
- The read method reads a single character, and returns a value (of type int) that corresponds to the character read
- Since the read method does not return the character itself, a type cast must be used:
- char next = (char)(readerObject.read());
- Unlike the Scanner class, the class BufferedReader has no methods to read a number from a text file
- A number must be read in as a string, and then converted to a value of the appropriate numeric type using one of the wrapper classes
- To read in a single number on a line by itself, first use the method readLine, and then use Integer.parseInt, Double.parseDouble, etc. to convert the string into a number
- If there are multiple numbers on a line, StringTokenizer can be used to decompose the string into tokens, and then the tokens can be converted as described above
- The method readLine of the class BufferedReader returns null when it tries to read beyond the end of a text file
- The method read of the class BufferedReader returns -1 when it tries to read beyond the end of a text file

BufferedReader

- When a file name is used as an argument to a constructor for opening a file, it is assumed that the file is in the same directory or folder as the one in which the program is run
- If it is not in the same directory, the full or relative path name must be given
- A path name not only gives the name of the file, but also the directory or folder in which the file exists
- A full path name gives a complete path name, starting from the root directory
- A relative path name gives the path to the file, starting with the directory in which the program is located
- The standard streams System.in, System.out, and System.err are automatically available to every Java program
- System.out is used for normal screen output
- System.err is used to output error messages to the screen

Binary Files

- Binary files store data in the same format used by computer memory to store the values of variables
- Java binary files, unlike other binary language files, are portable
- A binary file created by a Java program can be moved from one computer to another
- These files can then be read by a Java program, but only by a Java program
- An ObjectOutputStream object is created and connected to a binary file as follows:
- ObjectOutputStream outputStreamName = new ObjectOutputStream(new FileOutputStream(FileName));
- After opening the file, ObjectOutputStream methods can be used to write to the file
- Methods used to output primitive values include writeInt, writeDouble, writeChar, and writeBoolean
- UTF is an encoding scheme used to encode Unicode characters that favors the ASCII character set
- The method writeUTF can be used to output values of type String
- The stream should always be closed after writing
- The class ObjectInputStream is a stream class that can be used to read from a binary file
- An ObjectInputStream object is created and connected to a binary file as follows:
- ObjectInputStream inStreamName = new ObjectInputStream(new FileInputStream(FileName));
- After opening the file, ObjectInputStream methods can be used to read to the file
- Methods used to input primitive values include readInt, readDouble, readChar, and readBoolean
- The method readUTF is used to input values of type String
- If the file contains multiple types, each item type must be read in exactly the same order it was written to the file

Binary Files

- The stream should be closed after reading
- All of the ObjectInputStream methods that read from a binary file throw an EOFException when trying to read beyond the end of a file
- Objects can also be input and output from a binary file
- Use the writeObject method of the class ObjectOutputStream to write an object to a binary file
- Use the readObject method of the class ObjectInputStream to read an object from a binary file
- It is best to store the data of only one class type in any one file
- In addition, the class of the object being read or written must implement the Serializable interface
- Since an array is an object, arrays can also be read and written to binary files using readObject and writeObject
- The streams for sequential access to files are the ones most commonly used for file access in Java
- However, some applications require very rapid access to records in very large databases
- These applications need to have random access to particular parts of a file
- The stream class RandomAccessFile, which is in the java.io package, provides both read and write random access to a file in Java
- A random access file consists of a sequence of numbered bytes
- Although a random access file is byte oriented, there are methods that allow for reading or writing values of the primitive types as well as string values to/from a random access file