

Chp1 statements looping and branching

Code Magnets

```
class Shuffle1 {  
    public static void main(String[] args) {  
        int x = 3; while (x > 0) {  
            if (x > 2) {  
                System.out.print("a");  
            }  
            x = x - 1;  
            System.out.print("-");  
            if (x == 2) {  
                System.out.print("b c");  
            } if (x == 1) {  
                System.out.print("d"); x = x - 1;  
            } } } }
```

Be the compiler

```
class Exercise1a {  
    public static void main(  
        String [] args) {  
        int x = 1;  
        while ( x < 10 ) {  
            x = x + 1;  
            if ( x > 3) {  
                System.out.println("big x");  
            }  
        }  
    }  
}
```

This will compile and run (no output), but without a line added to the program, it would run forever in an infinite while loop!

```
class Exercise1b {  
    public static void main(  

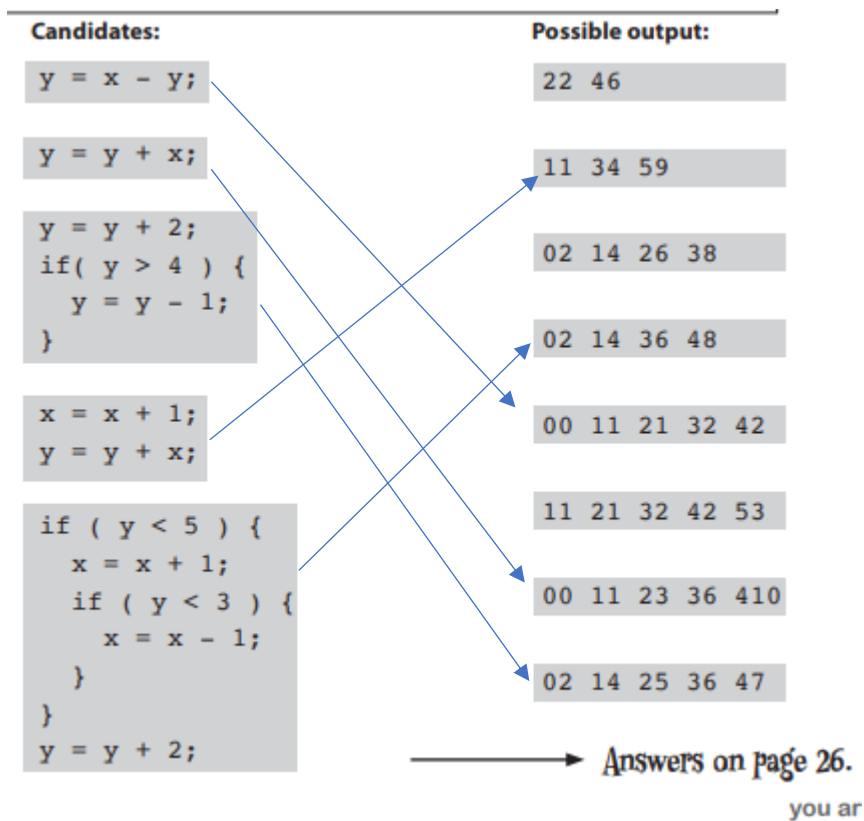
```

```
String [] args) {  
    int x = 5;  
    while ( x > 1 ) {  
        x = x - 1;  
        if ( x < 3 ) {  
            System.out.println("small x");  
        }  
    }  
}
```

This file won't compile without a class declaration, and don't forget } the matching curly brace!

```
lass Exercise1c {  
    public static void main(String [] args) {  
        int x = 5; while ( x > 1 ) {  
            x = x - 1;  
            if ( x < 3 ) {  
                System.out.println("small x");  
            }  
        }  
    }  
}
```

The while loop code must be inside a method. It can't just be hanging out inside the class.



Pool puzzle

```
class PoolPuzzleOne {
```

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

```
int x = 0;
```

```
while ( x < 4 ) {
```

```
System.out.print("a");
```

```
if ( x < 1 ) {
```

```
System.out.print(" ");
```

```
}
```

```
System.out.print("\n");
```

```
if ( x > 1 ) {
```

```
System.out.print(" oyster");
```

```
x = x + 2;
```

```
}
```

```
if ( x == 1 ) {
```

```

System.out.print("noys");
}
if ( x < 1 ) {
System.out.print("oise");
}
System.out.println();
x = x + 1;
}
}
}

```

Chp2 Classes and objects

Be the compiler A

```

class StreamingSong {
String title;
String artist;
int duration;
void play() {
System.out.println("Playing song");
}
void printDetails() {
System.out.println("This is + title + 17 by " + artist);
}
}

class StreamingSongTestDrive {
public static void main(String[] args) {
StreamingSong song = new StreamingSong();
song.artist = "The Beatles";
song.title = "Come Together";
song.play();
song.printDetails();
}
}

```

```
}
```

Be the compiler B

```
class Episode {  
    int seriesNumber  
    int episodeNumber;  
    void play() {  
        System.out.println("Playing episode " + episode Number);  
    }  
    void skipIntro() {  
        System.out.println("Skipping intro...");  
    }  
    void skipToNext () {  
        System.out.println("Loading next episode...");  
    }  
    class EpisodeTestDrive {  
        public static void main(String[] args) {  
            Episode episode new Episode();  
            episode.seriesNumber = 4;  
            episode.play();  
            episode.skipIntro();  
        }  
    }  
}
```

Code Magnets

```
class DrumKit {  
    boolean topHat = true;  
    boolean snare true;  
    void playTopHat() {  
        System.out.println("ding ding da-ding");  
    }  
    void playSnare () {  
        System.out.println("bang bang ba-bang");  
    }  
}
```

```

}
}
}
class DrumKitTestDrive {
public static void main(String[] args) {
    DrumKit d = new DrumKit();
    d.playSnare();
    d.snare = false;
    d.playTopHat();
    if (d.snare == true) {
        d.playSnare();
    }
}
}
}

```

Chp3 Primitives and references

BE the Compiler A

```

class Books {
    String title;
    String author;
}

class BooksTestDrive {
public static void main(String[] args){
    Books [] myBooks = new Books [3];
    int x = 0;
    myBooks[x] = new Books();
    myBooks[0] = new Books();
    myBooks[1] = new Books();
    myBooks[2] = new Books();
    myBooks [0].title = "The Grapes of Java";
    myBooks [1].title "The Java Gatsby";
    myBooks [2].title = "The Java Cookbook";
}
}

```

```

myBooks [0].author = "bob";
myBooks [1].author = "sue";
myBooks [2].author = "ian";
while (x < 3) {
    System.out.print (myBooks [x].title);
    System.out.print(" by ");
    System.out.println(myBooks [x].author);
    x = x + 1
}
}
}

```

BE the Compiler B

```

class Hobbits (
    String name;
    public static void main(String[] args) {
        Hobbits [] h = new Hobbits [3];
        int z = - 1;
        while (z < 2) {
            z = z + 1;
            h[z] =new Hobbits();
            h[z].name = "bilbo";
            if ( z ==1) {
                h[z].name "frodo";
            }
            if ( z ==2) {
                h[z].name = "sam";
            }
            System.out.print(h[z].name + " is a ");
            System.out.println("good Hobbit name");
        }
    }
}

```

}

```
1. int x = 34.5; ✗  
2. boolean boo = x; ✗  
3. int g = 17; ✓  
4. int y = g; ✓  
5. y = y + 10; ✓  
6. short s; ✓  
7. s = y; ✗  
8. byte b = 3; ✓  
9. byte v = b; ✓  
10. short n = 12; ✓  
11. v = n; ✗  
12. byte k = 128; ✗
```

Code Magnets

```
class TestArrays (  
    public static void main(String[] args) {  
        int[] index = new int[4];  
        index [0] = 1;  
        index [1] 3;  
        index [2] 0;  
        index [3] 2;  
    }  
}  
  
String[] islands = new String [4];  
islands [0] "Bermuda";  
islands [1] = "Fiji";  
islands [2] = "Azores";  
islands [3] = "Cozumel";  
  
int y = 0  
int ref;  
while (y < 4) {
```



```

ref = index [y];


System.out.print("island = ");

System.out.println(islands [ref]);

y = y + 1;
}
}
}

```

Chp 4 Methods use instance variables



Sharpen your pencil

KEEP

←

RIGHT

What's legal?

Given the method below, which of the method calls listed on the right are legal?

Put a checkmark next to the ones that are legal. (Some statements are there to assign values used in the method calls.)

```

int calcArea(int height, int width) {
    return height * width;
}

```

```

int a = calcArea(7, 12);
short c = 7;
calcArea(c, 15); ✓

int d = calcArea(57);

calcArea(2, 3); ✓

long t = 42;
int f = calcArea(t, 17);

int g = calcArea();

calcArea();

byte h = calcArea(4, 20);

int j = calcArea(2, 3, 5);

```

—————→ **Answers on page 93.**

Be the compiler

```

class Clock {
    String time;

    void setTime (String t) {
        time = t;
    }

    String getTime() {
        return time;
    }
}


class ClockTestDrive {

```

```

public static void main(String[] args) {
    Clock c = new Clock(); c.setTime("1245");
    String tod = c.getTime();
    System.out.println("time:+ tod);
}
}

```

	A class can have any number of these.	_____	Getter setter methods instance var
	A method can have only one of these.	_____	Return
	This can be implicitly promoted.	_____	Return argument
	I prefer my instance variables private.	_____	Encapsulation
	It really means "make a copy."	_____	Pass by value
	Only setters should update these.	_____	Instance variables
	A method can have many of these.	_____	Argument
	I return something by definition.	_____	Getter
	I shouldn't be used with instance variables.	_____	Public
	I can have many arguments.	_____	Method
	By definition, I take one argument.	_____	Setter
	These help create encapsulation.	_____	Getter setter public private
	I always fly solo.	_____	Return

Chp 5 Writing a program

```

class Output {
    public static void main(String[] args) {
        Output output = new Output();
        output.go();
    }
    void go() {
        int value = 7;
        for (int i = 1; i < 8; i++) {
            value++;
            if (i > 4) {

```

```

System.out.print(++value + "); "
}
if (value > 14) {
System.out.println(" i = + i);
break;
}
}
}
}
}
}

```

Code Magnets

```

class MultiFor {
public static void main(String[] args) {
for (int i = 0 ; i < 4 i++) {
for (int j = 4 ; j > 2 ; j --) {
System.out.println(i + " " + j);
}
if ( i ==1) {
i++;
}
}
}
}

```

Chp 6 Get to know the Java API

```

import java.util.ArrayList;

public class ArrayListMagnet {
public static void main(String[] args) {
ArrayList<String> a new ArrayList<String>();
a.add(0, "zero");
a.add(1, "one");
a.add(2, "two");
a.add (3, "three");
printList(a);
}
}

```

```

if (a.contains ("three")) {
a.add("four");
}
a.remove(2);
printList(a);
if (a.indexOf("four") != 4) {
a.add (4, "4.2");
}
printList(a);
if (a.contains("two")) { a.add("2.2");
}
printList(a);
}

public static void printList (ArrayList<String> list) {
for (String element list) {
System.out.print(element + " ");
}
System.out.println();
}
}

```

Chp 7 Inheritance and Polymorphism

	Output:
<div style="display: inline-block; vertical-align: middle;"> b.m1 () ; c.m2 () ; a.m3 () ; </div>	<div style="display: inline-block; vertical-align: middle;"> A's m1, A's m2, C's m3, 6 B's m1, A's m2, A's m3, A's m1, B's m2, A's m3, </div>
<div style="display: inline-block; vertical-align: middle;"> c.m1 () ; c.m2 () ; c.m3 () ; </div>	<div style="display: inline-block; vertical-align: middle;"> B's m1, A's m2, C's m3, 13 B's m1, C's m2, A's m3, </div>
<div style="display: inline-block; vertical-align: middle;"> a.m1 () ; b.m2 () ; c.m3 () ; </div>	<div style="display: inline-block; vertical-align: middle;"> B's m1, A's m2, C's m3, 6 A's m1, A's m2, C's m3, 13 </div>
<div style="display: inline-block; vertical-align: middle;"> a2.m1 () ; a2.m2 () ; a2.m3 () ; </div>	

Be the compiler

Set 1 will compile correctly.

Set 2 will not compile because Vampire's return type (`int`) is incompatible with the overridden method in `Monster`. The Vampire's `frighten()` method is not a valid override or overload of `Monster`'s `frighten()` method. Changing only the return type is insufficient for a valid overload, and since `int` is not compatible with `boolean`, it is not a valid override. Remember, if you change only the return type, it must be to a type compatible with the superclass version's return type for it to be considered an override.

Sets 3 and 4 will compile but produce the output:

arrrgh

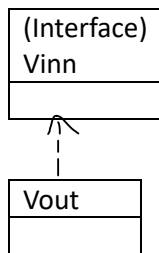
breathe fire

arrrgh

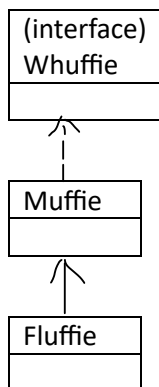
Chp 8 Interfaces and abstract classes

What's the pic

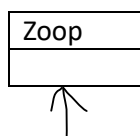
2.

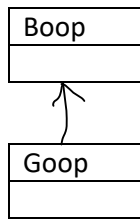


3.

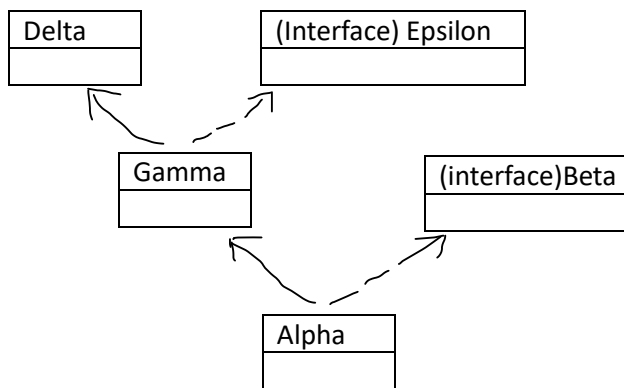


4.





5.



Chp 9 Constructors and garbage collection

1	<code>copyGC = null;</code>	No - This line attempts to access a variable that is out of scope.
2	<code>gc2 = null;</code>	Correct - gc2 was the sole reference variable to that object.
3	<code>newGC = gc3;</code>	Incorrect - This is another variable that is out of scope.
4	<code>gc1 = null;</code>	Correct - gc1 was the only reference because newGC is out of scope.
5	<code>newGC = null;</code>	Incorrect - newGC is out of scope.
6	<code>gc4 = null;</code>	Incorrect - gc3 still references that object.
7	<code>gc3 = gc2;</code>	Incorrect - gc4 still references that object.
8	<code>gc1 = gc4;</code>	Correct - Reassigning the only reference to that object.
9	<code>gc3 = null;</code>	Incorrect - gc4 still references that object.

Puzzle Objects

The Honey object initially assigned to the honeyPot variable is the most frequently referenced object in this class. However, it could be a bit more challenging to realize that every variable referencing the Honey object points to the same instance! Before the main() method finishes, there are 12 active references to this object. The kit.honeyPot variable remains valid for a while, but kit is set to null at

the end. Nonetheless, since raccoon.rk still references the Kit object, raccoon.kit.honeyPot (though never explicitly declared) also points to the Honey object.

Chp 10 Numbers and statics

Be the compiler

StaticSuper is a constructor and needs parentheses in its signature. As shown in the output below, the static blocks for both classes execute before their respective constructors.

- | | |
|---|---|
| 1. To use the Math class, the first step is to make an instance of it. | F |
| 2. You can mark a constructor with the static keyword. | F |
| 3. Static methods don't have access to instance variable state of the "this" object. | T |
| 4. It is good practice to call a static method using a reference variable. | F |
| 5. Static variables could be used to count the instances of a class. | T |
| 6. Constructors are called before static variables are initialized. | F |
| 7. MAX_SIZE would be a good name for a static final variable. | T |
| 8. A static initializer block runs before a class's constructor runs. | T |
| 9. If a class is marked final, all of its methods must be marked final. | F |
| 10. A final method can be overridden only if its class is extended. | F |
| 11. There is no wrapper class for boolean primitives. | F |
| 12. A wrapper is used when you want to treat a primitive like an object. | T |
| 13. The parseXxx methods always return a String. | F |
| 14. Formatting classes (which are decoupled from I/O) are in the java.format package. | F |

Chp 11 Collections and generics

sd

Given the following compilable statement:

```
Collections.sort(myArrayList);
```

- | | | |
|---|-------|--------------------------|
| 1. What must the class of the objects stored in <code>myArrayList</code> implement? | _____ | Comparable |
| 2. What method must the class of the objects stored in <code>myArrayList</code> implement? | _____ | <code>compareTo()</code> |
| 3. Can the class of the objects stored in <code>myArrayList</code> implement both <code>Comparator</code> AND <code>Comparable</code> ? | _____ | yes |

Given the following compilable statements (they both do the same thing):

```
Collections.sort(myArrayList, myCompare);  
myArrayList.sort(myCompare);
```

- | | | |
|---|-------|-------------------------|
| 4. Can the class of the objects stored in <code>myArrayList</code> implement <code>Comparable</code> ? | _____ | yes |
| 5. Can the class of the objects stored in <code>myArrayList</code> implement <code>Comparator</code> ? | _____ | yes |
| 6. Must the class of the objects stored in <code>myArrayList</code> implement <code>Comparable</code> ? | _____ | no |
| 7. Must the class of the objects stored in <code>myArrayList</code> implement <code>Comparator</code> ? | _____ | no |
| 8. What must the class of the <code>myCompare</code> object implement? | _____ | <code>Comparator</code> |
| 9. What method must the class of the <code>myCompare</code> object implement? | _____ | <code>compare()</code> |

Reverse Engineer

```
import java.util.*;  
  
public class SortMountains {  
  
    public static void main(String[] args) { new SortMountains().go();  
  
    public void go() {  
  
        List<Mountain> mountains = new ArrayList<>();  
        mountains.add(new Mountain ("Longs", 14255));  
        mountains.add(new Mountain ("Elbert", 14433));  
        mountains.add(new Mountain ("Maroon", 14156));  
        mountains.add(new Mountain ("Castle", 14265));  
        System.out.println("as entered:\n" + mountains);  
  
        mountains.sort((mount1, mount2) -> mount1.name.compareTo(mount2.name));  
        System.out.println("by name:\n" + mountains);  
  
        mountains.sort((mount1, mount2) -> mount2.height - mount1.height);  
        System.out.println("by height:\n" + mountains);  
  
    }  
  
    }  
  
    class Mountain { String name; int height;  
  
        Mountain(String name, int height) {  
  
            this.name = name;
```



```

this.height = height;
}

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

```

Sorting with lamdas

```

songList.sort((one, two) -> one.getBpm() - two.getBpm());

songList.sort((one, two) -> two.getTitle().compareTo(one.getTitle()));

```

tree set

1. It compiles
2. It throws exception
3. Make Book implement Comparable or can pass the TreeSet as a Comparator.

Compiles?

- ☒ `takeAnimals(new ArrayList<Animal>());`
- ☐ `takeDogs(new ArrayList<Animal>());`
- ☐ `takeAnimals(new ArrayList<Dog>());`
- ☒ `takeDogs(new ArrayList<>());`
- ☒ `List<Dog> dogs = new ArrayList<>();`
`takeDogs(dogs);`
- ☒ `takeSomeAnimals(new ArrayList<Dog>());`
- ☒ `takeSomeAnimals(new ArrayList<>());`
- ☒ `takeSomeAnimals(new ArrayList<Animal>());`
- ☒ `List<Animal> animals = new ArrayList<>();`
`takeSomeAnimals(animals);`
- ☐ `List<Object> objects = new ArrayList<>();`
`takeObjects(objects);`
- ☐ `takeObjects(new ArrayList<Dog>());`
- ☒ `takeObjects(new ArrayList<Object>());`

Chp 12 lambdas and streams

Code Magnets

```

import java.util.*;

import java.util.stream.*;

public class CoffeeOrder {

```

```

public static void main(String[] args) {

List coffees = List.of("Cappuccino", "Americano", "Espresso", "Cortado", "Mocha", "Cappuccino",
"Flat White", "Latte");

List coffeesEndingInO = coffees.stream() .filter(s -> s.endsWith("o")) .sorted() .distinct()
.collect(Collectors.toList());

System.out.println(coffeesEndingInO);


}

}

```

Check the box if the statement would compile.

- ☒ `Runnable r = () -> System.out.println("Hi!");`
- ☒ `Consumer<String> c = s -> System.out.println(s);`
- ☐ `Supplier<String> s = () -> System.out.println("Some string");`
- ☐ `Consumer<String> c = (s1, s2) -> System.out.println(s1 + s2);`
- ☐ `Runnable r = (String str) -> System.out.println(str);`
- ☒ `Function<String, Integer> f = s -> s.length();`
- ☒ `Supplier<String> s = () -> "Some string";`
- ☐ `Consumer<String> c = s -> "String" + s;`
- ☐ `Function<String, Integer> f = (int i) -> "i = " + i;`
- ☐ `Supplier<String> s = s -> "Some string: " + s;`
- ☐ `Function<String, Integer> f = (String s) -> s.length();`

 Sharpen your pencil

Which of these interfaces has a Single Abstract Method and can therefore be implemented as a lambda expression?

Interface	Modifier and Type	Method
BiPredicate	default	<code>BiPredicate<T,U> and(BiPredicate<? super T,? super U> other)</code>
	default	<code>BiPredicate<T,U> negate()</code>
	default	<code>BiPredicate<T,U> or(BiPredicate<? super T,? super U> other)</code>
	boolean	<code>test(T t, U u)</code>
ActionListener	void	<code>actionPerformed(ActionEvent e)</code>
Function	default	<code><V> Function<T,V> andThen(Function<? super R,? extends V> after)</code>
	R	<code>apply(T t)</code>
	default	<code><V> Function<V,R> compose(Function<? super V,? extends T> before)</code>
	static	<code><T> Function<T,T> identity()</code>
Iterator	default	<code>void forEachRemaining(Consumer<? super E> action)</code>
	boolean	<code>hasNext()</code>
	E	<code>next()</code>
	default	<code>void remove()</code>
SocketOption	String	<code>name()</code>
	Class<T>	<code>type()</code>

Chp 13 Exception Handling

TRUE OR FALSE

1. A try block must be followed by a catch and a finally block.
 2. If you write a method that might cause a compiler-checked exception, you must wrap that risky code in a try/catch block.
 3. Catch blocks can be polymorphic.
 4. Only "compiler checked" exceptions can be caught.
 5. If you define a try/catch block, a matching finally block is optional.
 6. If you define a try block, you can pair it with a matching catch or finally block, or both.
 7. If you write a method that declares that it can throw a compiler-checked exception, you must also wrap the exception throwing code in a try/catch block.
 8. The main() method in your program must handle all unhandled exceptions thrown to it.
 9. A single try block can have many different catch blocks.
 10. A method can throw only one kind of exception.
 11. A finally block will run regardless of whether an exception is thrown.
 12. A finally block can exist without a try block.
 13. A try block can exist by itself, without a catch block or a finally block.
 14. Handling an exception is sometimes referred to as "ducking."
 15. The order of catch blocks never matters.
 16. A method with a try block and a finally block can optionally declare a checked exception.
 17. Runtime exceptions must be handled or declared.
-
1. False, either or both are possible.
 2. False, you can specify the exception.
 3. True.
 4. False, runtime exceptions can be caught.
 5. True.
 6. True, both methods are valid.
 7. False, just declaring it is enough.
 8. False, but if it doesn't, the JVM might terminate.
 9. True.
 10. False.
 11. True, it's often used for cleaning up incomplete tasks.
 12. False.
 13. False.
 14. False, ducking is the same as declaring.

15. False, the broadest exceptions should be caught by the final catch blocks.

16. False, if there's no catch block, you must declare.

17. False.

Code Magnets

```
class MyEx extends Exception {  
    }  
  
public class ExTestDrive {  
    public static void main(String[] args) {  
        String test = args[0];  
        try {  
            System.out.print("t");  
            doRisky (test);  
            System.out.print("o");  
        } catch (MyEx e) {  
            System.out.print("a");  
        }  
        finally {  
            System.out.print("w");  
        }  
        System.out.println("s");  
    }  
  
    static void doRisky (String t) throws MyEx { System.out.print("h");  
        if ("yes".equals(t)) {  
            throw new MyEx(); }  
        System.out.print("r");  
    }  
}
```

Chp 14 Getting gui

Who I am



I got the whole GUI, in my hands.

Every event type has one of these.

The listener's key method.

This method gives JFrame its size.

You add code to this method but never call it.

When the user actually does something, it's an ____ .

Most of these are event sources.

I carry data back to the listener.

An addXxxListener() method says an object is an ____ .

How a listener signs up.

The method where all the graphics code goes.

I'm typically bound to an instance.

The "g" in (Graphics g) is really of this class.

The method that gets paintComponent() rolling.

The package where most of the Swingers reside.

yu

- _____ JFrame
- _____ Listener interface
- _____ actionPerformed()
- _____ setSize()
- _____ paintComponent()
- _____ event
- _____ swing components
- _____ event object
- _____ event source
- _____ addXxxListener()
- _____ paintComponent()
- _____ inner class
- _____ Graphics2D
- _____ repaint
- _____ javax.swing

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

class InnerButton {
    private JButton button;
```



```
    public static void main(String[] args) {
        InnerButton gui = new InnerButton();
        gui.go();
    }
```

```
    public void go() {
        JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(
            JFrame.EXIT_ON_CLOSE);
```

```
        button = new JButton("A");
        button.addActionListener();
```

(new ButtoListener())

```
        frame.getContentPane().add(
            BorderLayout.SOUTH, button);
        frame.setSize(200, 100);
        frame.setVisible(true);
```

```
    }

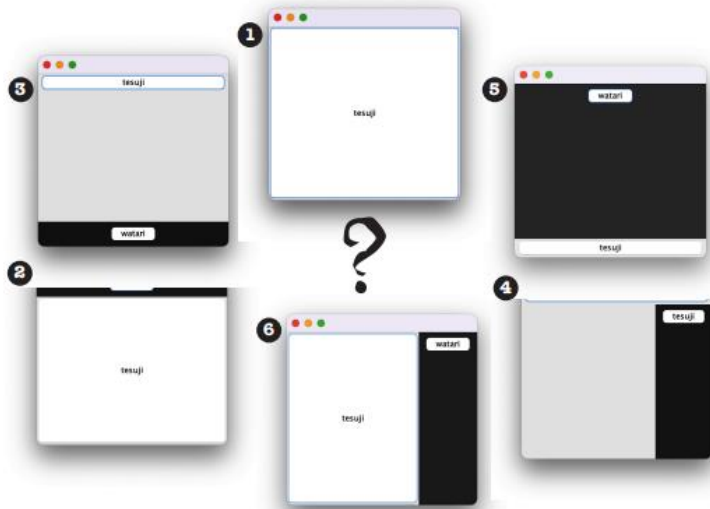
    class ButtonListener extends ActionListener {
        public void actionPerformed(ActionEvent e) {
            if (button.getText().equals("A")) {
                button.setText("B");
            } else {
                button.setText("A");
            }
        }
    }
}
```

implements

After correcting this code, it will generate a GUI with a button that switches between A and B when clicked. The `addActionListener()` method accepts a class that implements the ActionListener interface. ActionListener is an interface, and interfaces are implemented, not extended.

Chp 15 Using swing

Each of the five code fragments will use the layout that fragment would produce.



A `JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(button);
frame.getContentPane().add(BorderLayout.NORTH, buttonTwo);
frame.getContentPane().add(BorderLayout.EAST, panel);`

B `JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER, button);
frame.getContentPane().add(BorderLayout.EAST, panel);`

C `JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER, button);`

D `JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.NORTH, panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.CENTER, button);`

E `JFrame frame = new JFrame();
JPanel panel = new JPanel();
panel.setBackground(Color.darkGray);
JButton button = new JButton("tesuji");
JButton buttonTwo = new JButton("watari");
frame.getContentPane().add(BorderLayout.SOUTH, panel);
panel.add(buttonTwo);
frame.getContentPane().add(BorderLayout.NORTH, button);`

1 → C

2 → D

3 → E

4 → A

6 → B

Chp 16 Serialization and file I/O

- | | |
|--|---|
| 1. Serialization is appropriate when saving data for non-Java programs to use. | F |
| 2. Object state can be saved only by using serialization. | F |
| 3. ObjectOutputStream is a class used to save serializable objects. | T |
| 4. Chain streams can be used on their own or with connection streams. | F |
| 5. A single call to writeObject() can cause many objects to be saved. | T |
| 6. All classes are serializable by default. | F |
| 7. The java.nio.file.Path class can be used to locate files. | F |
| 8. If a superclass is not serializable, then the subclass can't be serializable. | F |
| 9. Only classes that implement AutoCloseable can be used in try-with-resources statements. | T |
| 10. When an object is deserialized, its constructor does not run. | T |
| 11. Both serialization and saving to a text file can throw exceptions. | T |
| 12. BufferedWriters can be chained to FileWriters. | T |
| 13. File objects represent files, but not directories. | F |
| 14. You can't force a buffer to send its data before it's full. | F |
| 15. Both file readers and file writers can optionally be buffered. | T |
| 16. The methods on the Files class let you operate on files and directories. | T |
| 17. Try-with-resources statements cannot include explicit finally blocks. | F |

Chp 17 Networking and threads

Who I am?

1. I need to be shut down or I might live forever → Thread
2. I let you talk to a remote machine → Socket
3. I might be thrown by sleep() and await() → InterruptedException
4. If you want to reuse Threads, you should use me → ThreadPoolExecutor
5. You need to know me if you want to connect to another machine → Socket
6. I'm like a separate process running on the machine → Thread
7. I can give you the ExecutorService you need → Executors
8. You need one of me if you want clients to connect to me → ServerSocket
9. I can help you make your multithreaded code more predictable → ExecutorService
10. I represent a job to run → Runnable
11. I store the IP address and port of the server → InetSocketAddress

Code Magnets

```
import java.io.*;

import java.net.InetSocketAddress;

import java.nio.channels.*;

import java.time.format.FormatStyle;

import java.util.concurrent.TimeUnit;

import static java.nio.charset.StandardCharsets.UTF_8;

import static java.time.LocalDateTime.now;

import static java.time.format.DateTimeFormatter.ofLocalizedTime;

public class PingingClient {

    public static void main(String[] args) {

        InetSocketAddress server = new InetSocketAddress("127.0.0.1", 5000);

        try (SocketChannel channel = SocketChannel.open(server)) {

            PrintWriter writer = new PrintWriter(Channels.newWriter(channel, UTF_8));

            System.out.println("Networking established");

            for (int i = 0; i < 10; i++) {

                String message = "ping " + i;

                writer.println(message);

                writer.flush();

                String currentTime = now().format(ofLocalizedTime(FormatStyle.MEDIUM));

                System.out.println(currentTime + " Sent " + message);

                TimeUnit.SECONDS.sleep(1);

            }

        } catch (IOException | InterruptedException e) {

            e.printStackTrace();

        }

    }

}
```

Network Connection: Ensured that the `SocketChannel` is properly managed with a try-with-resources statement to automatically close the channel.

Exception Handling: Combined the exception handling for `IOException` and `InterruptedException` in a single catch block for simplicity.

PrintWriter: Ensured that the `PrintWriter` uses the proper encoding by passing `UTF_8`.

Chp 18 race conditions and immutable data

The issue with the code is that it has a race condition where two threads are accessing and modifying the same data structure (letters in the `Data` class) concurrently. This leads to inconsistent and unpredictable output. To fix this problem and ensure the output is correct every time, we need to synchronize access to the shared resource. There are a couple of ways to do this: using the `synchronized` keyword or using concurrent data structures. Here's how to fix the code using the `synchronized` keyword:

```
import java.util.*;

import java.util.concurrent.*;

public class TwoThreadsWriting {

    public static void main(String[] args) {

        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        Data data = new Data();

        threadPool.execute(() -> addLetterToData('a', data));

        threadPool.execute(() -> addLetterToData('A', data));

        threadPool.shutdown();

    }

    private static void addLetterToData(char letter, Data data) {

        for (int i = 0; i < 26; i++) {

            data.addLetter(letter++);

            try {

                Thread.sleep(50);

            } catch (InterruptedException ignored) {}

        }

        System.out.println(Thread.currentThread().getName() + data.getLetters());

        System.out.println(Thread.currentThread().getName() + " size = " + data.getLetters().size());

    }

}
```

```
}  
  
final class Data {  
  
    private final List<String> letters = new ArrayList<>();  
  
    public synchronized List<String> getLetters() {  
  
        return new ArrayList<>(letters); // Return a copy to avoid concurrent modification issues  
  
    }  
  
    public synchronized void addLetter(char letter) {  
  
        letters.add(String.valueOf(letter));  
  
    }  
  
}
```