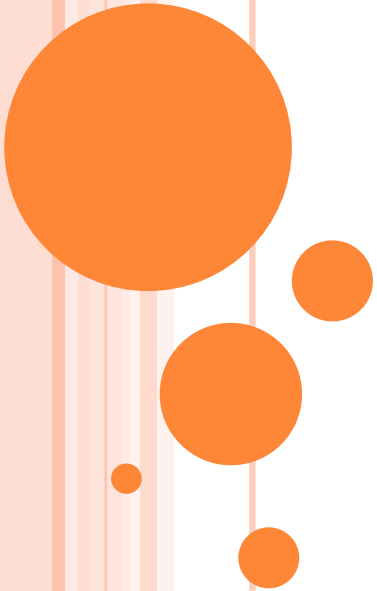
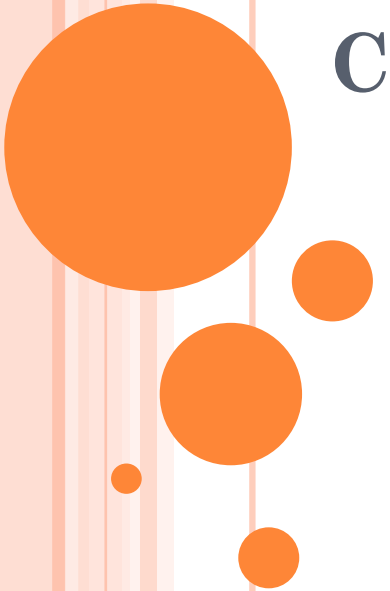
CO1109
Introduction to Java and Object-Oriented Programming



Dr Manolis Falelakis

m.falelakis@gold.ac.uk

SIM Revision 2019– Part II



Chapter 0: Outline

OUTLINE – DAY 2

1. Classes and objects
2. Object behaviour
3. Java Library
4. Variables
5. Inheritance
6. Numbers
7. File I/O
8. Searching in Arrays
9. Exceptions
10. Extras

OUTLINE – DAY 2

1.	Classes and objects	
2.	Object behaviour	
3.	Java Library	1b
4.	Variables	
5.	Inheritance	6a, 6b, 6c
6.	Numbers	
7.	File I/O	5a, 5b, 5c, 5d
8.	Searching in Arrays	
9.	Exceptions	1c, 3c
10.	Static and final	3b

Chapter 1: Object-Oriented Programming

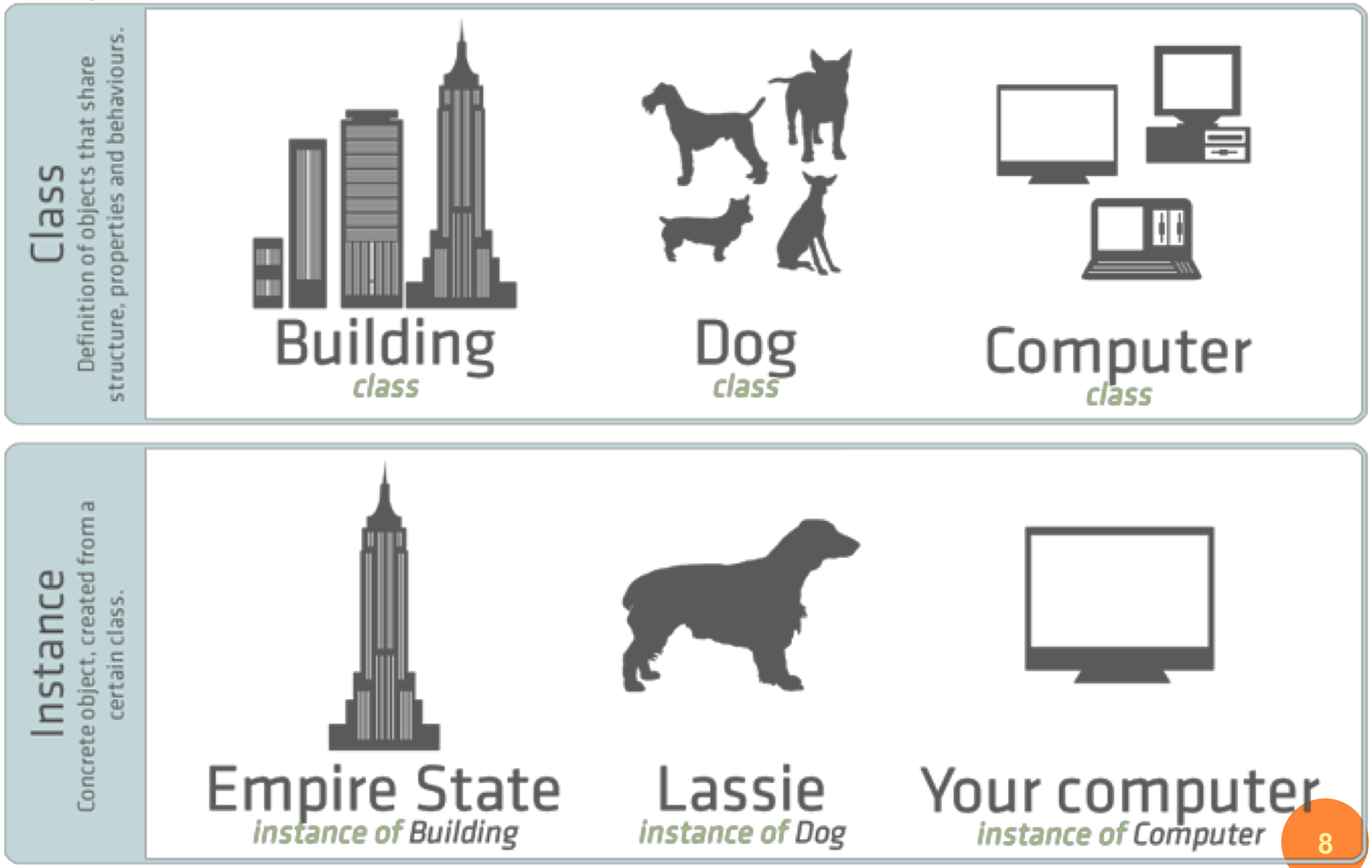
Object-Oriented Programming

- Java is an object-oriented programming language
- As the term implies, object is a fundamental entity in a Java program
- Objects can be used effectively to represent real-world entities
- For instance, an object might represent a particular employee in a company
- Objects are instances of Classes

What is a class?

- Not an object!
- A blueprint that defines the variables and methods common to all objects or instances of a particular type.
- Used to create an object instance!

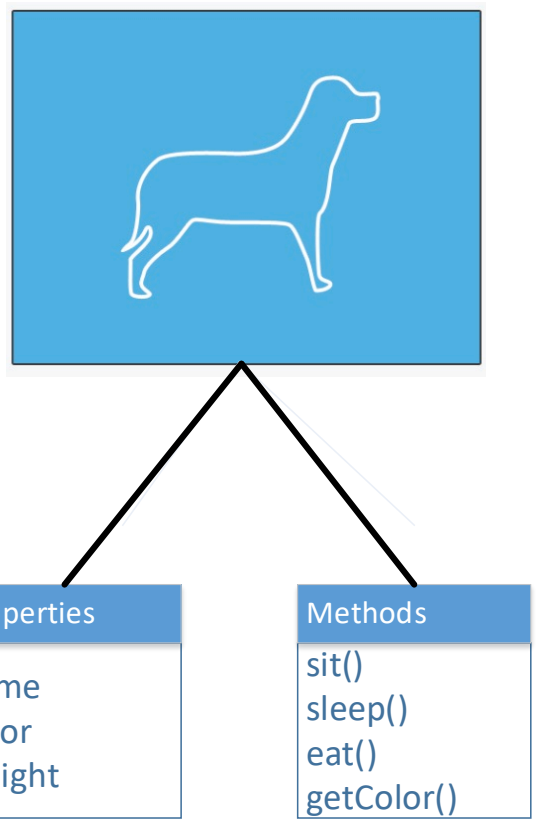
Class vs Instances



- **Variables**
- **Methods**

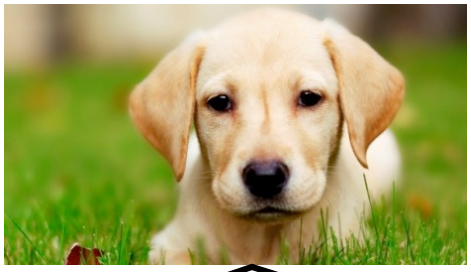
Class vs Instances

- A Class (the concept)



- **Variables**
- **Methods**

- Objects (the realisations)



Properties	Methods
name = Victor color = brown weight = 2kg	sit() - never sleep() - fast eat() - a little getColor()

Properties	Methods
name = Jack color = white weight = 10kg	sit() - fast sleep() - fast eat() - a lot getColor()

Instance variables

- a.k.a. fields, properties,(attributes)
- Things that an object knows about itself.
- State of the object
- Examples
  - weight
  - age
  - breed
  - ..

Methods

- Things an object can *DO*
  - Behaviour

- Examples:
  - sit();
  - sleep();
  - eat();
  - getColor();

What is inheritance?

- You know a lot about a class by knowing about its superclass

- Even if you don't know what a Marmot is, if I told you it was an Animal, you would know it could move, sleep and breathe.

- Inheritance is the OO term for this: defining classes in terms of other classes or the act of making one class inherit from another class.

Class Hierarchy

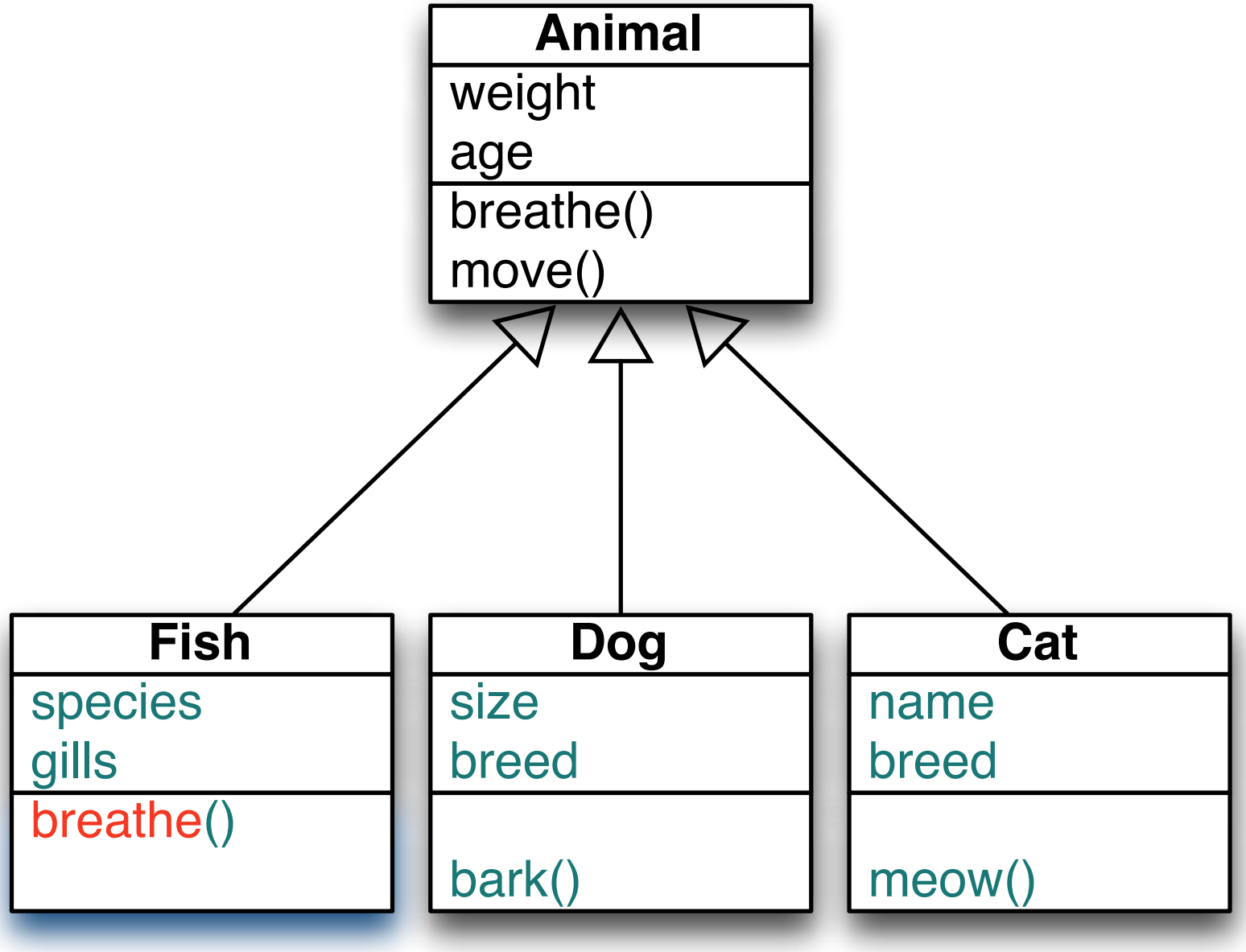
- Superclass (e.g. Animal)
  - The *parent* class
- Subclass (e.g. Dog, Fish)
  - The *child* class
  - **Inheriting** from the parent.

Class Hierarchy

- Subclasses may **add** variables and methods
  - A Dog might have a bark() method

- Subclasses may **override** methods to provide specialized implementations
  - A fish breathes but its breathe method would work much differently. So the Fish subclass should override the breathe() method to use gills.

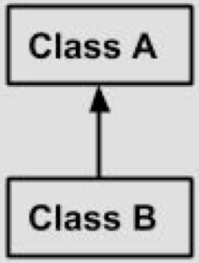
Class Hierarchy

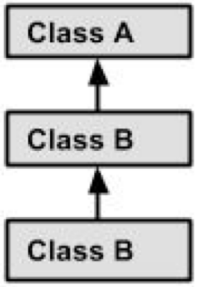
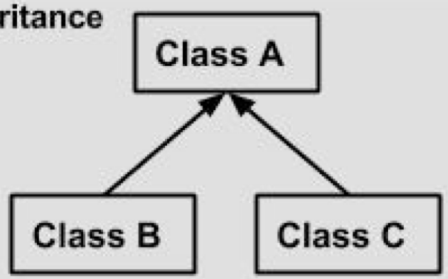


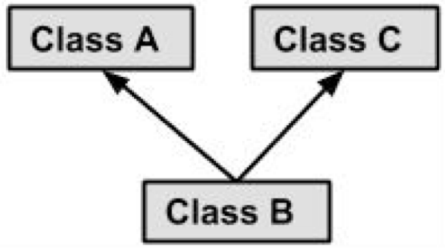
Class Hierarchy

- Not limited to one layer of inheritance
- One class can inherit from another which inherits from another…
- In Java, every object inherits from **`Object`**, either directly or indirectly

Class Hierarchy

Single Inheritance
 
	public class A {
	…….
	}
	public class B **extends** A {
	……..
	}

Multi Level Inheritance 	public class A { ……} public class B **extends** A {……} public class C **extends** B {…… }
Hierarchical Inheritance 	public class A { ……} public class B **extends** A {……} public class C **extends** A {…… }

Multiple Inheritance
 
	public class A { ……}
	public class B {……}
	public class C **extends** A,B {
	……
	} // Java does not support mutiple Inheritance

Benefits of Inheritance

- Software can be reused
- Objects only have to do what they really need to do, while deferring to a base class (superclass) for "common functionality"
- Importantly: natural, helps conceptual modeling!

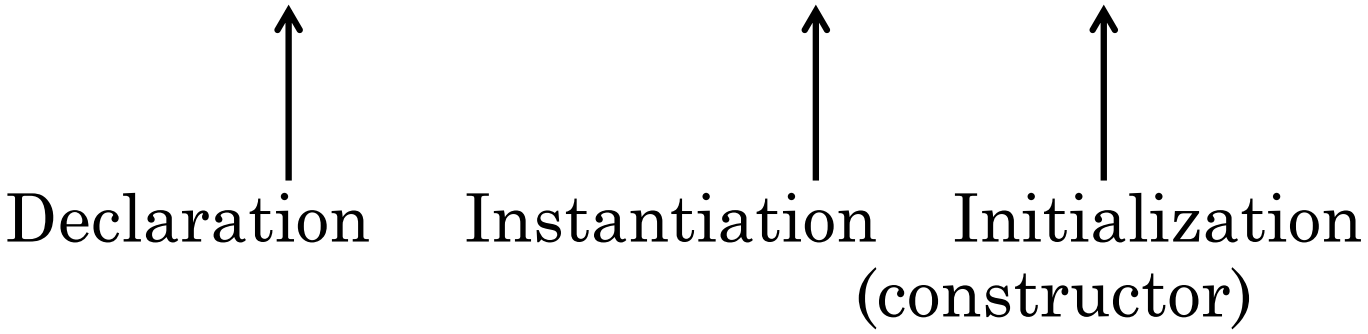
Polymorphism

- A method that performs some type of operation as part of different types of objects is said to be polymorphic.
  - animal.breathe();
  - fish.breathe();
  - dog.breathe();
- In Greek (Πολυμορφισμός) means ‘many forms’
- The property of an object’s behaviour to differ from its parent and siblings
- Modifying behaviour to suit itself

Creating Objects

- You create an Object with the *new* keyword

```
Point originOne = new Point(23, 94);
```



Constructor

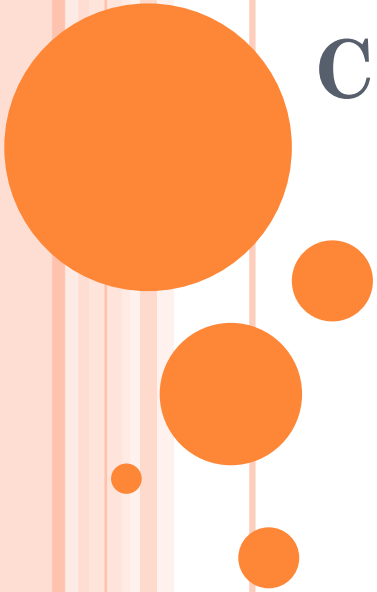
- a special method used to instantiate an object of a class
- run when you instantiate an object.
- has the same name as the class
- has no return type

Using a class

- Need a main() method.
  - A single one for each application/ class collection
  - entry point of any java program
- Use **`new`** to create an instance of a class.
- Access variables and methods with dot operator
- e.g.
  - `radio.playTune();`
  - `myTitle = book.title;`

NOTE - PUBLIC STATIC VOID MAIN

- `Public`
  - It has to be public so that java runtime can execute this method
- `static`
  - When JRE starts, there is no object of the class present.
  - → has to be static so that JVM can load the class into memory and call the main method.
- `void`
  - Once executed, program ends so no point to return anything!
- `main`
  - That's it (by convention)!
- `String[] args`
  - You can change the name of the argument!
    - `public static void main(String[] aaaaaa)`



Chapter 2: Object behaviour

Terminology

- **Instance**
  - A particular object
- **Instance variables**
  - Variables attached to the object
- **Instance methods**
  - Methods attached to the object (not the class)
- **Encapsulation**
  - Accomplished by access control on variables
  - An important concept of OO Programming

API

- stands for Application Programming Interface
- The API for a class includes all its public methods and public variables.

Invoking a method: using its API

- `bicycle.changeGear(lowGear)`

- Three parts
  - The object to address (bicycle)
  - The part of the bicycle to address (changeGear method)
  - The information the method needs to operate (lowGear)

Encapsulation

- Definition:
  - the creation of self-contained modules that contain both the data and the processing.

- The term *encapsulation* is often used interchangeably with information hiding
  - Make instance variables private
  - Provide ***getter*** and ***setter*** methods.

Benefits of encapsulation

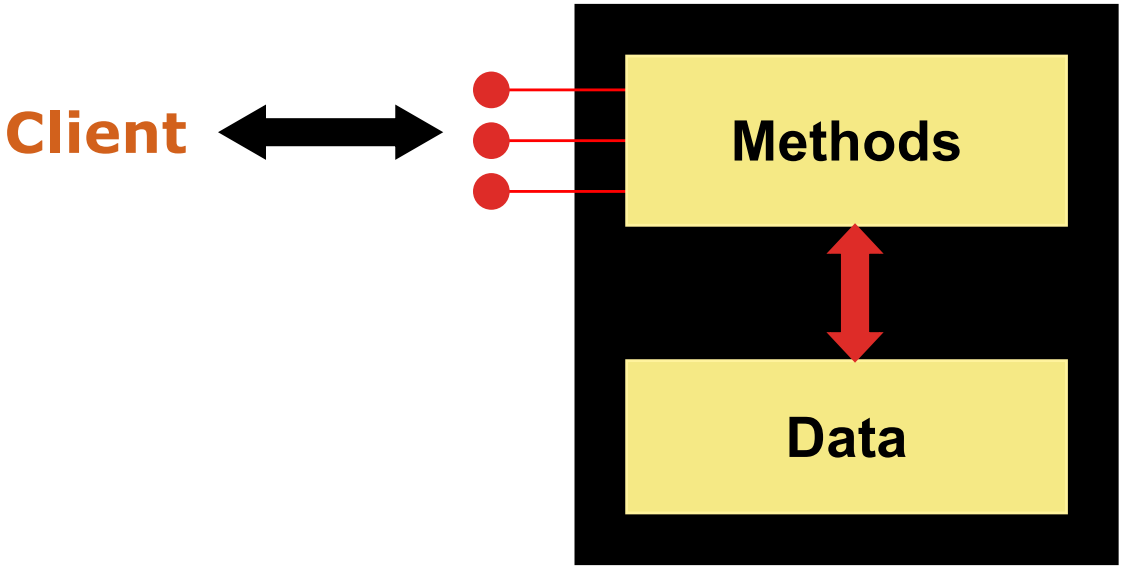
- **Information hiding**
  - An object has a public interface that other objects use to communicate with it
  - An object can have private state, which can be accessed and possibly modified by methods
  - You don't have to understand an object's inner workings to utilize it

→Modularity

- The code can be written and maintained independently of other objects
- Objects can be reused in different applications

Encapsulation

- An encapsulated object can be thought of as a *black box* -- its inner workings are hidden from the client
- The client invokes the interface methods of the object, which manages the instance data



Getters and Setters

- Often called accessors and mutators
- Use them to access or modify private variables.

- Example:

```
private float weight;

public float getWeight()
{
   return weight;
}
```

```
public void setWeight(float wght)
{
     if (wght < 100.5)
    {
        weight = wght;
    }
}
```

EXERCISE 2.1

What is the output of the following code? Choose one of the options below.

```
public class CDemo{
    public static void main (String [] args){
        CDemo cd = new CDemo();
        cd.first().last().second();
    }

    CDemo first(){
        System.out.print("A");

        return this;
    }
```

EXERCISE 2.1

```
    CDemo second(){
        System.out.print ("B");
        return this;
    }

    CDemo last(){
        System.out.print ("C");
        return this;
    }
}
```

a. **ABC**
b. **ACB**
c. **BCA**
d. **The program will produce a compilation error**

Ans: b. (ACB)

EXERCISE 2.2

What is the output of the code given below when an object of type Confuse *is created and used to call method* startUp()*? Explain your answer.*

EXERCISE 2.2

```
class Confuse{
 private int x;
 private int y;
 public Confuse(){
        x = 3; y = 6;
 }
 private  void first(int z){
        x = z; y++;
 }
 private void second(int s, int
t){
        setXY( (y+t), (x-s));
        s = 1; t = 3;
 }
```

Answer: -1

x	y
3	6
6	7
13	-1
-1	0

```
 private void setXY( int s, int y){
        x = s;
        this.y = y;
        first( y);
 }
 private void display(){
        System.out.println(x + y);
 }
 public void startUp(){
        first( y);
        second( y, x);
        display();
 }
 public static void main (String[] args){
        Confuse c = new Confuse();
        c.startUp();
 }
}
```

Chapter 3: The Java Library

Java Libary

- Includes specifications for classes (and routines, data structures, variables).
- Also called the Java API
- Huge collection of available classes.
- Use it!

http://docs.oracle.com/javase/9/docs/api/

Import

- Defined after the package statement at the beginning of your class definition
- Example:
  - `java.util.ArrayList list = new java.util.ArrayList();`

  or
  - `import java.util.ArrayList;`
  - `ArrayList list = new ArrayList();`

- Better use full class name in imports for clarity
  - Good: `import java.util.ArrayList;`
  - Bad: `import java.util.*;`

The import Declaration

- All classes of the `java.lang` package are imported automatically into all programs
- It's as if all programs contain the following line:

```
import java.lang.*;
```

- That's why we didn't have to import the `System` or `String` classes explicitly in earlier programs
- The `Scanner` class, on the other hand, is part of the `java.util` package, and therefore must be imported

The Random Class

- The `Random` class is part of the `java.util` package
- It provides methods that generate pseudorandom numbers
- A `Random` object performs complicated calculations based on a *seed value* to produce a stream of seemingly random values

The Math Class

- The `Math` class is part of the `java.lang` package
- The `Math` class contains methods that perform various mathematical functions
- These include:
  - absolute value
  - square root
  - exponentiation
  - trigonometric functions

The Math Class

- The methods of the `Math` class are *static methods* (also called *class methods*)
- Static methods can be invoked through the class name – no object of the `Math` class is needed

```
value = Math.cos(90) + Math.sqrt(delta);
```

Lists

- **list**:
  - a collection storing an ordered sequence of elements
    - each element is accessible by a 0-based **index**
    - a list has a **size** (number of elements that have been added)
    - elements can be added to the front, back, or elsewhere
- In Java, a list can be represented as an **ArrayList** object

Idea of a List

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)
  - `[]`
  - You can add items to the list.
    - The default behavior is to add to the end of the list.
  - The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
- Think of an "array list" as an automatically resizing array object.
- Internally, the list is implemented using an array and a size field.

USEFUL ARRAYLIST METHODS

- `add(obj)`
  - // Appends obj to the end of list
- `add(index,obj)`
  - // inserts given value just before the given index, shifting subsequent values to the right
- `clear()`
  - // removes all elements of the list
- `indexOf(obj)`
  - // returns first index where given object is found in list (-1 if not found)
- `get(index)`
  - // returns the value at given index
- `remove(index)`
  - // removes/returns object at given index, shifting subsequent values to the left
- `set(index, value)`
  - // replaces object at given index with given value
- `size()`
  - // returns the number of elements in list
- `toString()`
  - // returns a string representation of the list such as "[3, 42, -7, 15]"

Type Parameters (Generics)

```
ArrayList<Type> name = new ArrayList<Type>();
```

- When constructing an ArrayList, you must specify the type of elements it will contain between < and >.
  - This is called a *type parameter* or a *generic* class.
  - Allows the same ArrayList class to store lists of different types.

```
ArrayList<String> names = new ArrayList<String>();
names.add("Joe");
names.add("Mary");
```

Array vs ArrayList

- Construction

```
String[] names = new String[5];
ArrayList<String> list = new ArrayList<String>();
```

- storing a value

```
names[0] = "Jessica";
list.add("Jessica");
```

- retrieving a value

```
String s = names[0];
String s = list.get(0);
```

ArrayList vs Vector

Their API is fairly similar. However they differ wrt:

- Synchronisation
  - Vectors are thread-safe
  - ArrayLists are not

  →if a thread is working on a Vector, no other thread can get hold of it.
- Performance
  - ArrayLists are usually faster
    - Due to not being thread-safe
- Data growth
  - When runs out of room, a Vector is doubled in size.
  - An ArrayList is expanded by 50%

- In practice ArrayLists are preferred!

NOTE: LENGTH VS LENGTH( ) VS SIZE( )

- `length`
  - a final variable, returns the storing capacity of an **array**.
  - not the number of elements stored.
- `length()`
  - a method of String objects
  - returns the number of characters present in the string.
- `size()`
  - a method of Collection classes (e.g. `ArrayList`)
  - returns the number of elements present.
  - defined in `java.util.Collection` interface.

NOTE: LENGTH VS LENGTH( ) VS SIZE( )

Check out the following valid examples:

- `String str;`
  - `str.length();` returns the number of characters of str.
- `int[] arr;`
  - `arr.length;` returns arr's capacity
- `String[] args;`
  - `args.length;` returns args' capacity.
  - `args[0].length();` returns args[0] number of characters
- `ArrayList<String> al;`
  - `al.size();` returns al's size
  - `al.get(0).length();` returns number of characters in al's first element

EXERCISE 3.3

Consider an array of Integer objects, declared as:

```
Integer[] array = new Integer[num];
```

where num is an int variable declared and initialized elsewhere. Assume also that array[] contains Integer objects, declared elsewhere in our program.

Write a loop that prints the integer value of each element, using:

- a while loop
- a for loop with a counter int i

EXERCISE 3.3

In a different version of our program, we choose to store our Integer objects in an ArrayList instead of an array, using the command:

```
ArrayList<Integer> a = new ArrayList<Integer>(num);
```

Assuming the ArrayList `a` is initialized elsewhere, write a loop that prints out the integer value of each element.

EXERCISE 3.3

Ans

```
for (int i=0; i <a.size();i++){
      System.out.println(a.get(i));
}
```

With enhanced for (“for each”)

```
for (Integer i : a){
      System.out.println(i);
}
```

EXERCISE 3.6

Write a method with signature

`static double average Vector<Double> v`

which returns the average of all elements in a non-empty Vector of Doubles.

Ans:

```
public static double average (Vector<Double> v){
        double result =0;
        for(int i=0; i<v.size(); i++){
                result+=v.get(i);
        }
        return result / (double)v.size();
}
```

Exercise 3.7

Write a method with heading

`static int product(Vector v)`

which returns the product of all elements in a non-empty Vector of Integers.

```
public static int product (Vector<Integer> v ){
        int result =1;
        for(int i=0; i<v.size(); i++){
                result*=v.get(i);// because of autoboxing!
        }
        return result;
}
```

EXERCISE 3.8

The Java library provides the static method Math.random() that generates a random number between zero and one. Using this or any other method

(i) Provide a Java expression that returns an integer between 1 and 5

Ans: (int)(Math.random()*5 + 1)

(ii) Write a method with signature

`int getRandomElement(int[] arr)`

that randomly selects an element of arr and returns its index (position in arr)

```
public int getRandomElement(int[] arr){
        int result = (int)(Math.random()*arr.length);
        return result;
}
```

EXERCISE 3.8

(iii) Write a program that initializes an array of ints, calls getRandomElement 1000 times and then for each array element it prints out the number of times it has been selected.

```
public class RandomQuestion {
  public static void main(String[] args){
    RandomQuestion rq = new RandomQuestion();
    int[] arr = {1, 2, 3, 4, 5};
    int[] results = new int[arr.length];
    for(int i=0; i<1000; i++)
      results[rq.getRandomElement(arr)]++;
    for(int i=0; i<arr.length; i++)
      System.out.println("Element "+arr[i]+" chosen
"+results[i]+" times");
  }
//getRandomElement(int[] arr){
}
```

EXERCISE 3.9

The following class will not compile. Can you identify why and suggest a fix?

```
import java.util.*;
public class Q2a {
    public static void main(String[] args) {
        ArrayList<String> arrList = new ArrayList<String>();
        String[] items = { "One", "Two", "Three", "Four",
"Five" };
        for(String str: items){
            arrList.add(str);
        }
        int size = items.size();
        System.out.println(size);
    }
}
```


Ans: items.size()->items.length

EXERCISE 3.10

(i) Write a method *addStars* that accepts an array list of strings as a parameter and places a * after each element.

Example: if an array list named `list` initially stores:

```
[the, quick, brown, fox]
```

Then the call of `addStars(list);` makes it store:

```
[the, *, quick, *, brown, *, fox, *]
```

Answer

```
public static void addStars(ArrayList<String>
list) {
      for (int i = 1; i <= list.size(); i += 2){
         list.add(i, "*");
    }
}
```

EXERCISE 3.10

(ii) Write a method *removeStars* that accepts an array list of strings, assuming that every other element is a *, and removes the stars (undoing what was done by *addStars* above).

Answer

```
public void removeStars(ArrayList<String>
list){
      for (int i = 1; i < list.size(); i++) {
            list.remove(i);
      }
}
```

Mock exam 2019: Question 1b

For each of the following expressions say whether they type check correctly or not, by writing in your answer book "yes" or "no", respectively.

A. `Math.abs("hello".size());`

B. `Math.abs("hello".length);`

C. `Math.abs("hello".length());`

D. `"two".compareTo(3);`

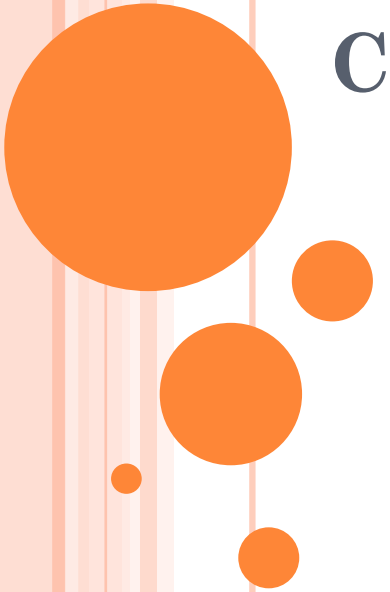
E. `int a = "one".compareTo("two")+11;`

F. `Float.parseFloat("12");`

G. `Float.parseFloat("12.0");`

H. `Math.abs(Float.parseFloat("11.1")+"two".compareTo("1"));`

Ans: No, No, Yes, No, Yes, Yes, Yes, Yes



Chapter 4: Variables

Terminology

- *Variables* – data placeholders
- *Declaration* – specifying a name and a type in the form *type name*
- *Scope* – where the variable is visible to be used by name in a program, determined by its declaration.

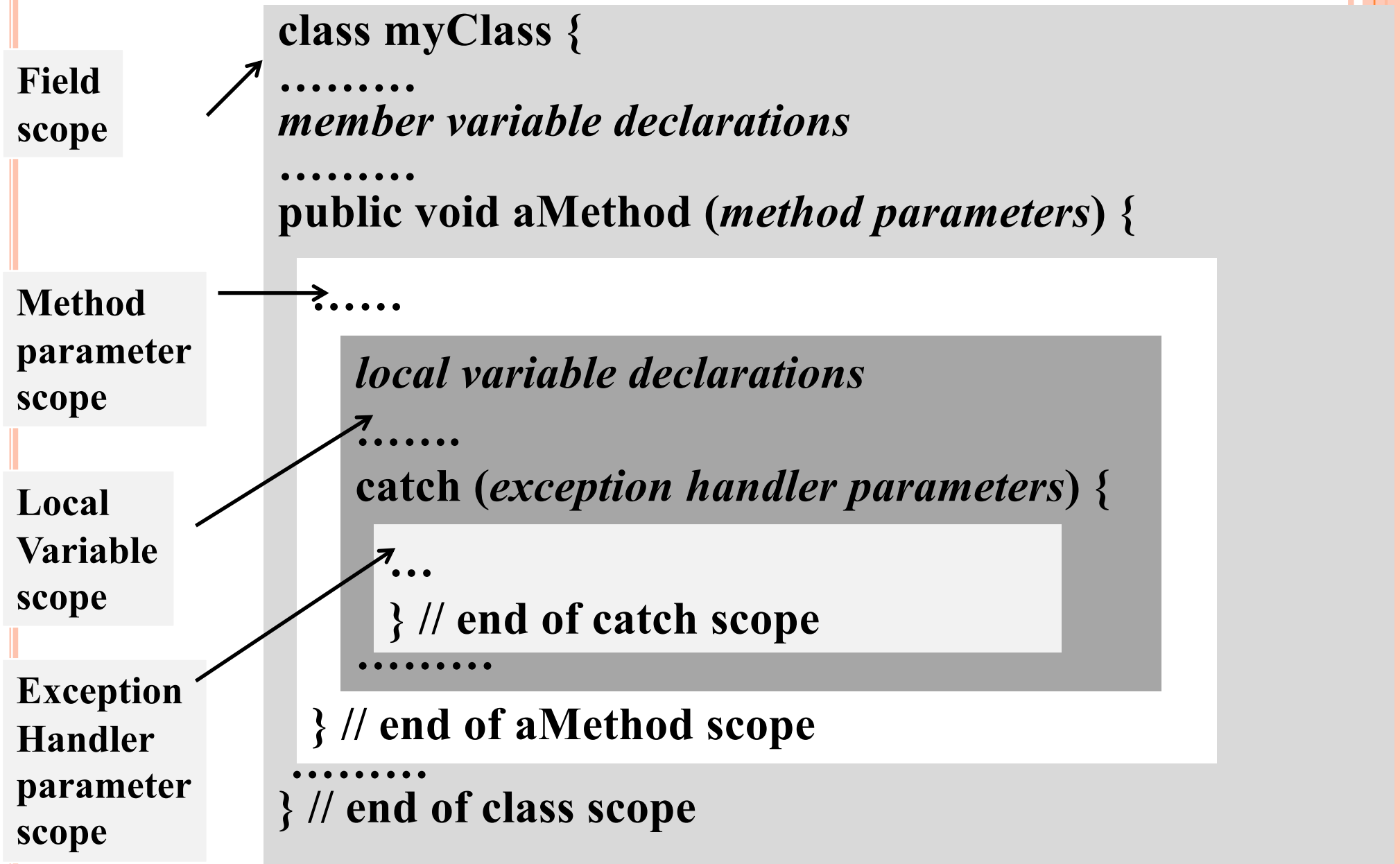
Declaration examples

```
int i = 5;

float f = 2.0;
double d = 55.13;

char c = 'S';
boolean b = true;
```

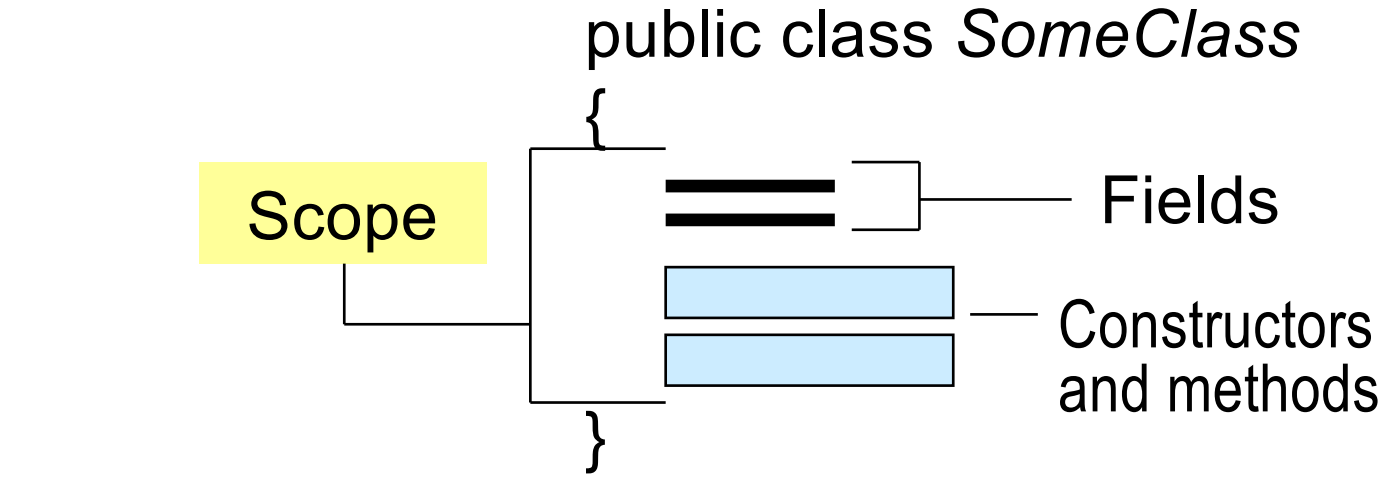
Scope - Instance vs Local Variable

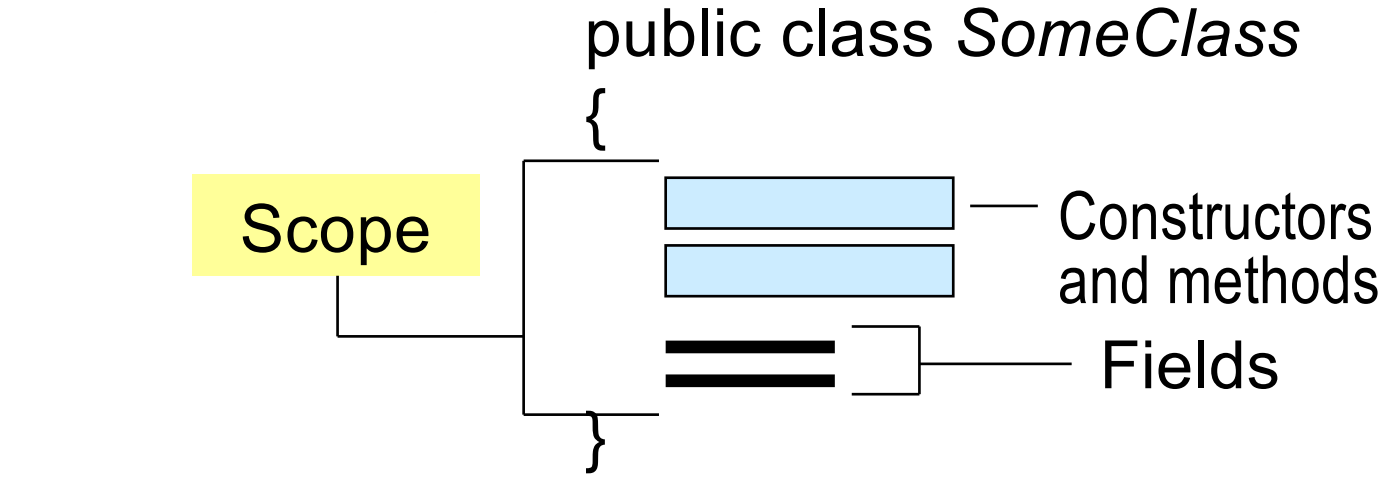
- **Instance variables (fields)** are declared outside all constructors and methods.
- The scope of an instance variable is the whole class.
- **Local variables** are declared inside a constructor or a method.
- The scope of a local variable is from its declaration down to the closing brace of the block in which it is declared.

VISUALISING SCOPE

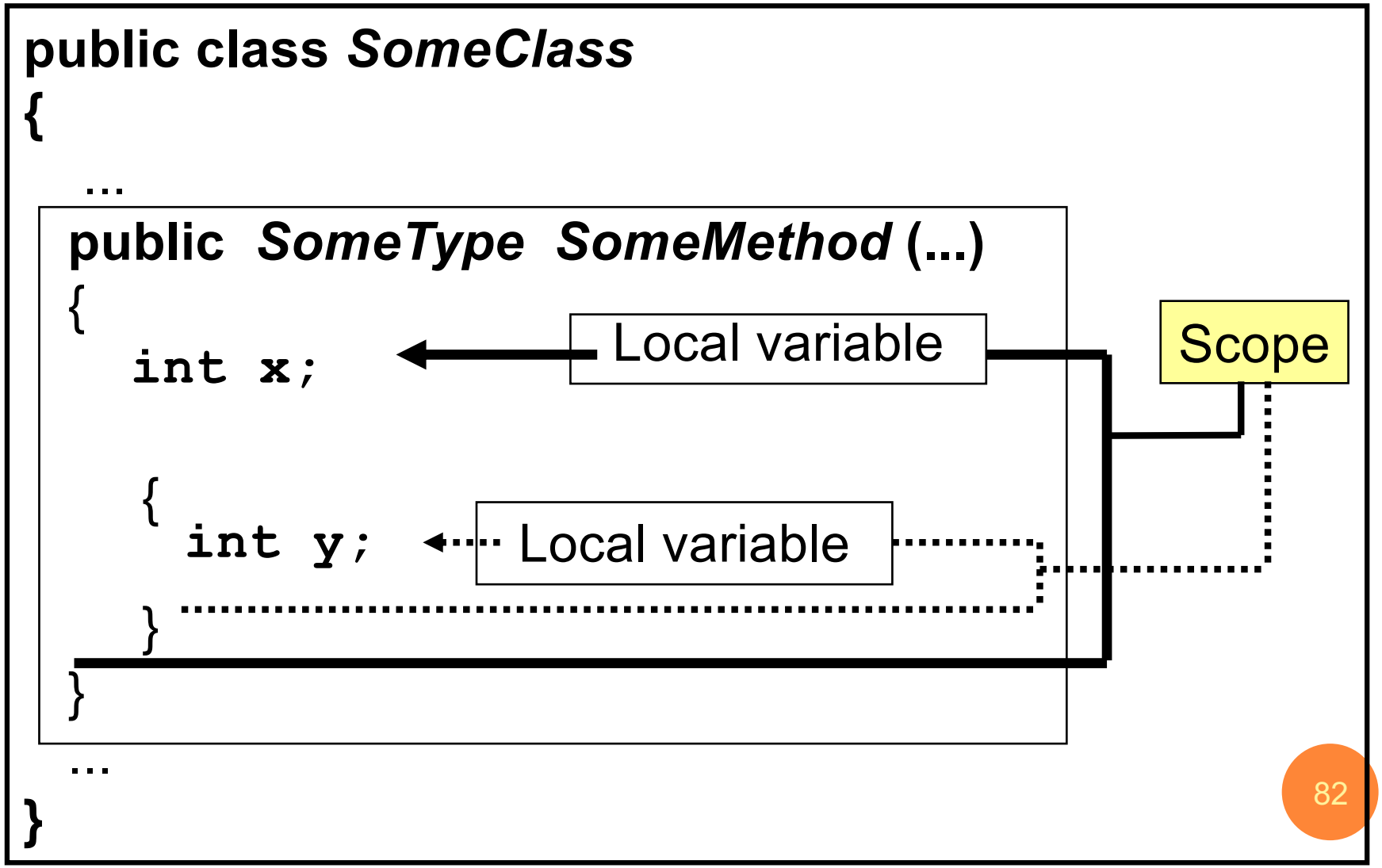


INSTANCE VARIABLES





Local Variables



Common mistakes with Variables

- Common mistakes:

```
public void SomeMethod (...)
{
   int x;
   ...
   int   x = 5;   // should be:  x = 5;
   ...
```

Variable declared twice — syntax error

Common mistakes with Variables

- Common mistakes:

```
private int cubeX;
...
public SomeClass(...)       // constructor
{
   int cubeX = 5;   // should be: cubeX = 5;
   ...
```

A field is hidden by a local variable; the value of the field cubeX remains unset

Data types

- Java has two categories of data types
  - Primitive
  - Reference

Example: Swap Numbers

```
public class SwapNumbers {
	public static void swap(int a, int b){
		int temp=a;
		a=b;
		b=temp;
	}

	public static void main(String []args){
		int x=5;
		int y=3;
		swap(x,y);
		System.out.println("x:" + x + " y:" + y);
	}
}
```

Example: Swap Numbers

- What is the output of the class SwapNumbers?

- Ans:
  - x:5, y:3

Example: Swap Persons

```
public class SwapPersons {
	public static void swap(String p1, String p2){
		String temp = p2;
		p2 = p1;
		p1 = temp;
		}

	public static void main(String []args){
		String s1 = new String("Alice");
		String s2 = new String("Bob");
		swap(s1,s2);
		System.out.println("s1:" + s1 + " s2:" + s2);
	}
}
```

Example: Swap Persons

- What is the output of the class SwapPersons?

- Ans:
  - s1:Alice s2:Bob

  - Strings are immutable!

Example: Swap Persons Array

```
public class SwapPersonsArray {
	public static void swap(String[] p){
		String temp = p[1];
		p[1]=p[0];
		p[0]= temp;
	}

	public static void main(String []args){
		String[] persons = {"Alice", "Bob"};
		System.out.println("Before - p1:" + persons[0]
				+ " p2:" + persons[1]);
		swap(persons);
		System.out.println("After - p1:" + persons[0]
				+ " p2:" + persons[1]);
	}
}
```

Example: Swap Persons Array

- What is the output of the class SwapPersonsArray?

- Ans:
  - Before - p1:Alice p2:Bob
  - After - p1:Bob p2:Alice

EXERCISE 4.1

What is the output of the following code?

```
class Call {
    static void method1 (int x){
        x = 0;
    }

    static void method2 (int [] x){
        x [0] = 0;
    }
```

EXERCISE 4.1

```
public static void main (String [] args){
      int x1 = 3;
      method1 (x1);
      System.out.println (x1);
      int [] x2 = { 3, 2, 4 };
      method2 (x2);
      for (int i = 0; i < x2.length; i++){
            System.out.print (x2 [i]);
            if (i != x2.length - 1)
                  System.out.print (", ");
            }
      }
}
```

Ans:

3

0, 2, 4

EXERCISE 4.2

What is the output of the following program?

```
class StringParams{
	public static void main(String [ ] args){
		String n="hello";
		p(n);
		System.out.println(n);
	}
	static void p(String m){
		m="goodbye";
	}
}
```

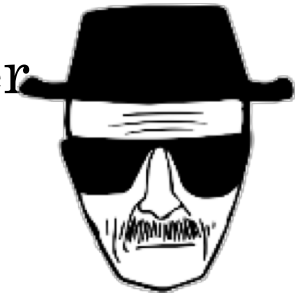

Answer: hello

EXERCISE 4.3

What is the output of the following program?

```
public class Danger {
    public static void main(String []args){
        String s1, s2 = "Heisenberg";

        s1 = s2;
        s2 = "Hawking";

        System.out.println(s1+ " is the danger");
       }
}
```

- Ans: Heisenberg is the danger



EXERCISE 4.4

What is the output of the following program?

```
public class Swap {
	public static void swap(int[] a){
		int temp = a[1];
		a[1] = a[0];
		a[0] = temp;
	}

	public static void main(String []args){
		int [] x = {5, 3};
		swap(x);
		System.out.println("x[0]:" + x[0] +
					"x[1]:" + x[1]);
	}
}
```

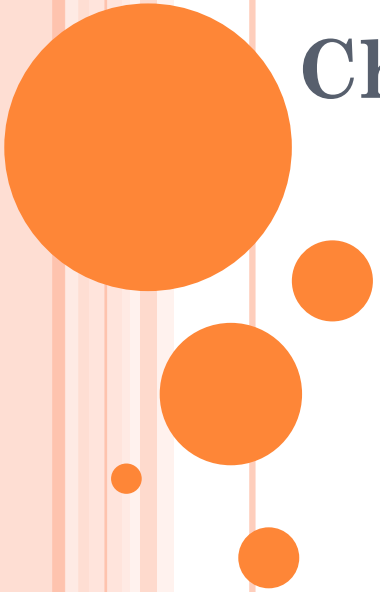
- Ans: x[0]:3 x[1]:5

EXERCISE 4.5

What is the output of the following program?

```
class A {
	static void method(int x)
	{
		x=3;
	}
	public static void main(String [ ] args)
	{
		int x;
		x=11;
		method(x);
		System.out.println(x);
	}
}
```

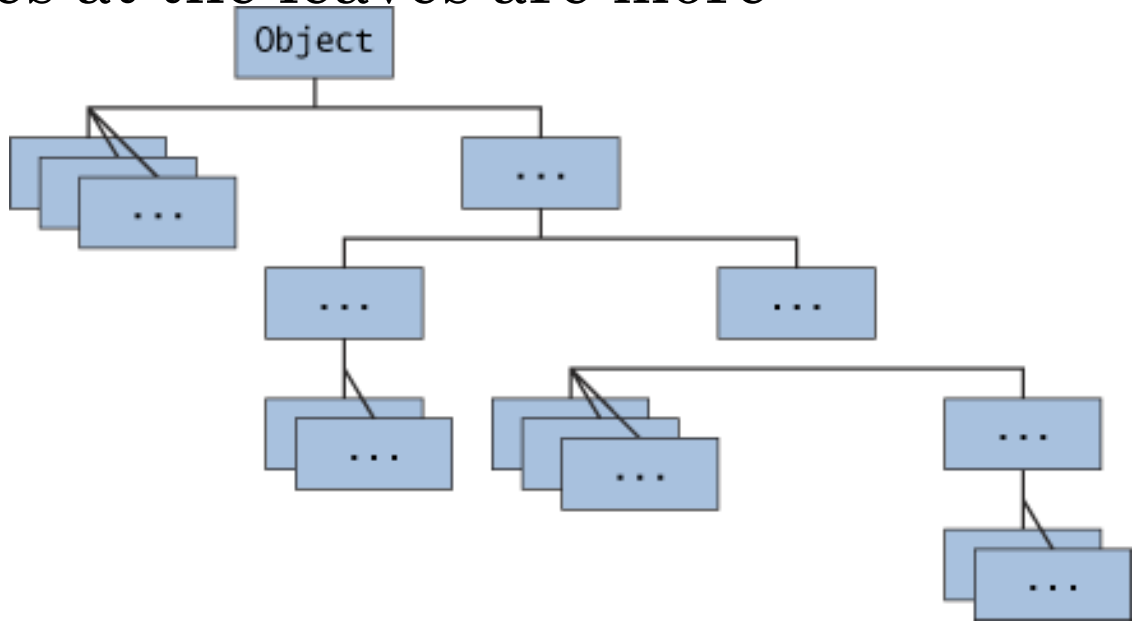
Ans: 11



Chapter 5: Inheritance

Inheritance

- Object-oriented programming allows classes to *inherit* commonly used state and behavior from other classes.
- All classes participate in inheritance, and descend from Object
- Classes towards the top are more general, whereas classes at the leaves are more specialized



Overriding vs overloading

- **Overriding** occurs when a subclass has an instance (i.e., non-static) method with the same name and parameters as a superclass.
  - Static methods are **hidden**, not overriden.
- **Overloading** occurs when a class has multiple methods of the same name, but with different parameter lists.
- Different return types make no difference

Using Super

- If your method overrides one of its superclass' methods, you can invoke the overridden method through the use of super
  - super.override()

EXAMPLE: WHAT WILL BE PRINTED?

```
public class Superclass {
    public boolean aVariable;

    public void aMethod()
    { aVariable = true; }
}

public class Subclass extends Superclass {

  public void aMethod() { //overrides aMethod
  in Superclass
        super.aMethod();
        System.out.println(aVariable);

    }
}
```

ANSWER

- The print statements in Subclass's aMethod display true

- The simple name aMethod refers to the one declared in Subclass, which overrides the one in Superclass.
  - → to refer to aMethod inherited from Superclass, Subclass use super.aMethod();

SUPER CONSTRUCTORS

- You can use the *super* keyword to invoke a constructor of the super class
- Must be the first line of a subclass constructor
- If you do not explicitly call a super constructor, the compiler will call the super's no-arg constructor automatically
  - (or throw an error if one does not exist)

CONSTRUCTORS VS METHODS (1)

- Constructors and methods differ in three aspects of the signature: (i) modifiers (ii) return type (iii) name.

- (i) Modifiers:
  - Like methods, constructors can have any of the access modifiers: public, protected, private, or none.
  - Unlike methods, constructors can **only** take access modifiers. Therefore, constructors cannot be abstract, final, static, or synchronized.

CONSTRUCTORS VS METHODS (2)

- (ii) Return Type:
  - Methods can have any valid return type (incl. void).
  - Constructors have no return type, not even void.

- (iii) Name:
  - Constructors have the same name as the class.
  - Methods must have a different name from that of the class.

CONSTRUCTOR OVERLOADING

You can have multiple constructors passing different arguments – overloaded constructors.

```
public class Point
{    public int x = 0;
     public int y = 0;
     //no-arg constructor
     public Point()
     {
     }

     //constructor with arguments, setting state
     public Point(int x, int y)
     {
          this.x = x;
          this.y = y;
     }
}
```

Calling another constructor

- A constructor can be called from within a different constructor
- To avoid duplicate code, often use
  - this()
  - super()
- Both have to be in the first line

  → Cannot use them together

Inheritance

- **Which method gets called**
  - Most specific
    - Lowest in hierarchy
  - Compiler guarantees this
  - A match must be found or an error is produced at compilation
- **One direction only**
  - A subclass can call a superclass
  - Super **cannot** call a sub.

Inheritance

- **Relationships: Extends**
  - Passes the "Is-A" test
  - A sub is-a super
  - A super is not a sub
    - Ex: a dog is an animal. An animal is not a dog (not always): `Dog extends Animal.`
- **Related but not inherited**
  - Other relationships, such as "Has-A"
  - One class can reference another and have a relationship that isn't inheritance
    - A PC has a monitor but is not a monitor!
    - A person has a chin, but it is not a chin (or the opposite)

Inheritance

- **What is inherited**
  - Public – inherited
  - Private – not inherited
  - If the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

Access levels – Table.

Modifier	Class	Package	Subclass	World
Public	Y	Y	Y	Y
Protected	Y	Y	Y	N
No Modifier	Y	Y	N	N
Private	Y	N	N	N

EXAMPLE: OVERLOADING

*Which ones of the following methods do **not** constitute a valid case for overloading?*

1.
```
public void print() { }
public void print(String msg) {}
```
2.
```
public void print(String m, int c) {}
public void print(int c, String m) {}
```
3.
```
public int print() { }
public void print(String msg) {}
```
4.
```
public int print() {}
public void print() {}
```

Answer: Case 4 (overloaded methods must have different arguments)

EXERCISE 5.3

Consider the following class definition:

```
class Base {
        Base(int i){}
}
class Extension extends Base {
Extension(int i){
        //Add code here
        }
}
```

Which one of the following lines can be placed in the commented line above without producing compilation errors?

1) super();
2) this();
3) this(99);
4) super(99);

Ans: 4) super(99);

EXERCISE 5.7

The following class will not compile. Can you identify why? Specify the lines of code that cause the problem:

```
class Problem {
        public static void main(String args[]){
                Problem p = new Problem();
                p.doSomething("11");
        }
        public int doSomething(int i){
        }
        public void doSomething(String s){
        }
        public void doSomething(int i){
        }
        public void doSomething(String s, int i){
        }
}
```

Ans: Improper method overloading: First and third method do not constitute a valid case because they have the same arguments, even if their return types are different.

EXERCISE 5.8

Which of the following is a true statement regarding a constructor?

(i) *It has no return type*
(ii) *Its return type changes based on how you write it*
(iii) *Its return type is void*
(iv) *Its return type is int*

Ans: (i)

EXERCISE 5.9

Consider the class Parent that follows. Define a class Child that extends Parent. Within it define a method that overrides doNothing and another method that overloads doNothing. Use appropriate comments to indicate which method is which.

```
public class Parent {
	public int doNothing(int i){
		return i;
	}
}
```

Ans:

```
class Child extends Parent {
	public int doNothing(int i){
		return i;
	}
	public void doNothing(String s){
	}
}
```

EXERCISE 5.10

For a class called Fraction that has two data members, an int numer and an int denom (for the numerator and denominator respectively), do the following:

(i) Write a constructor that accepts values for the numerator and denominator as arguments.

(ii) Write a toString method that returns a String representation of the fraction.

(iii) Write an equals method for fractions. Assume that all common factors have been removed from numerator and denominator.

EXERCISE 5.10

(i) Write a constructor that accepts values for the numerator and denominator as arguments.

```
public Fraction(int numerator, int denominator) {
        numer = numerator;
        denom = denominator;
}
```

EXERCISE 5.10

(ii) Write a toString method that returns a String representation of the fraction

```
Public String toString(){
      String ret = "" + numer + " / " + denom;
      return ret;
}
```

EXERCISE 5.10

(iii) Write an equals method for fractions. Assume that all common factors have been removed from numerator and denominator.

```
public boolean equals(Fraction other){
      Fraction f = other;
      if(numer == f.numer && denom == f.denom){
            return true;
      }
      return false;
}
```

EXERCISE 5.11

Define a Pet class, so that it includes:

- *two String fields; name and colour,*
- *a constructor with two parameters*
- *a suitable toString() method*

```
public class Pet {
	public static void main(String[] args){
		int i = 1;
		doTheMath(i);
	}
	public void doTheMath(int n){
		for(int i=0; i<n; i=i+2){
			System.out.println(i+2);
		}
	}
}
```

EXERCISE 5.11

Define a Pet class:

```
public class Pet {
	private String name;
	private String colour;

	public Pet(String n, String c){
		this.name=n;
		this.colour=c;
	}
	public String toString(){
		return ("Pet name: "+this.name+", colour:
"+this.colour);
	}
}
```

EXERCISE 5.11

Use inheritance, to define a Dog class. Include two fields: boolean barks and int numberOfFleas. Also include a constructor for Dog with two parameters and another one with no parameters.

```
public class Pet {
	private String name;
	private String colour;

	public Pet(String n, String c){
		this.name=n;
		this.colour=c;
	}
	public String toString(){
		return ("Pet name: "+this.name+", colour: "+this.colour);
	}
}
```

EXERCISE 5.11

```
class Dog extends Pet{
	private boolean barks;
	private int numberOfFleas;

	public Dog(boolean b, int n){
		super("Fred", "Black");
		this.barks= b;
		this.numberOfFleas = n;
	}
	public Dog(){
		super("Fred", "Black");
		this.barks= true;
		this.numberOfFleas = 100;
	}
}
```

Mock exam 2019: Question 6a

For each of the following statements, say whether they are TRUE or FALSE

A. In Java, fields are often referred to as instance variables.
B. The keyword new is used to create new classes.
C. An Instance method must have the same name as the name of its containing class.
D. Constructors always have a void return type.
E. Constructors cannot have parameters.
F. The compiler supplies a default constructor if no constructors are provided for a class.
G. When extending a class, you can overload instance methods inherited from a superclass by redefining methods with the same name.
H. When extending a class, all fields and methods are being inherited.

A.T
B.F
C.F
D.F
E.F
F.T
G.T
H.F

Mock exam 2019: Question 6b

Consider the following class Shape

```
public class Shape {
        private String color;
        public static void main(String[] args){
        }
}
```

Modify the class Shape so that it belongs to the geometry package and make sure that it has the following characteristics:

- *a method that returns the color ('getter' method).*
- *a method that sets the color ('setter' method).*
- *a constructor with no parameters that automatically sets color to "Red".*
- *an overloaded constructor that sets the color according to an input parameter.*
- *a printShape() method that prints all of its fields.*

MOCK EXAM 2019: QUESTION 6B

```
package geometry; //belongs to the geometry package

public class Shape {
        private String color;
        public static void main(String[] args){
        }
        public String getColor() { //getter method
                return color;
        }
        public void setColor(String color) { //setter method
                this.color = color;
        }
```

MOCK EXAM 2019: QUESTION 6B

```
package geometry; //belongs to the geometry package

        public Shape(){ //Default constructor
                setColor("Red");
        }
        public Shape(String col){ //Overloaded constructor
                setColor(col);
        }
        public void printShape(){ //printShape method
                System.out.println("color= "+this.color);
        }
```

Mock exam 2019: Question 6c

Extend the `Shape` *class to create the classes:* `Circle` *and* `Rectangle`*, making sure that they belong to the same package as their superclass. Write them so that they comply with the following:*

- *the Circle class includes an int field named radius while the Rectangle class includes two int fields named sideOne and sideTwo respectively*
- *write constructors for both Circle and Rectangle that set all their fields. To this end, use the super keyword as appropriate.*
- *In the Circle class, write a method called getArea() that calculates the surface area of the circle.*

Hint: the PI value can be found in the Math package

Mock exam 2019: Question 6c

```
public class Rectangle extends Shape {
        private int sideOne, sideTwo;
        public Rectangle(String col, int s1, int s2) {
                super(col);
                this.sideOne=s1;
                this.sideTwo=s2;
        }
}
```

Mock exam 2019: Question 6c

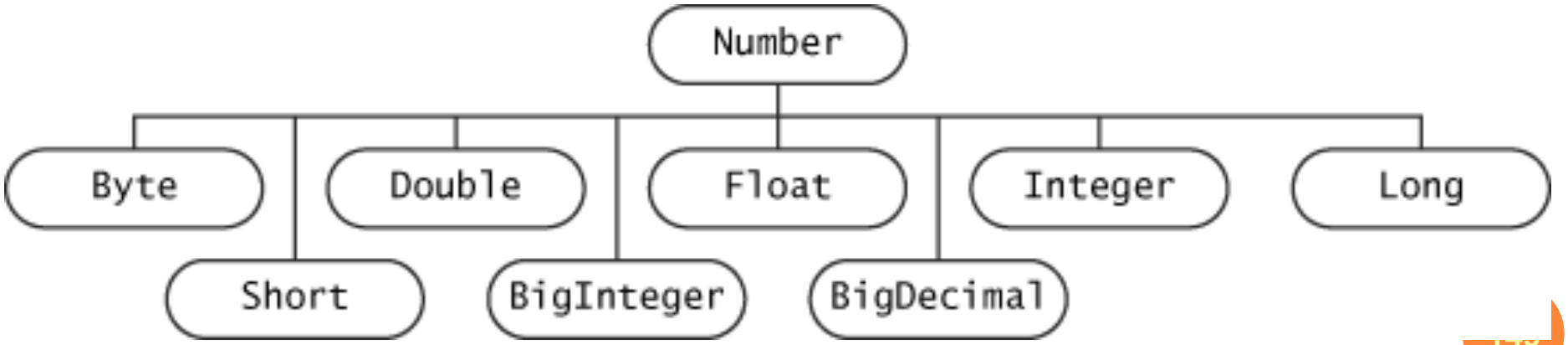
```
public class Circle extends Shape {
	private  int radius;

	public Circle(String col, int radius) {
		super(col);
		this.radius=radius;
	}
	public double getArea(){
		return Math.PI*Math.sqrt((double)this.radius);
	}
}
```

Chapter 6: Numbers

(not the most accurate title ever)

Numbers

- Two ways to use numbers
  - You can use the primitives
  - You can use the wrapper class equivalents



Wrapper Classes

- The `java.lang` package contains *wrapper classes* that correspond to every primitive type:

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`
`void`	`Void`

Wrapper Classes

- The following declaration creates an `Integer` object which represents the integer 40 as an object

```
Integer age = new Integer(40);
```

- An object of a wrapper class can be used in any situation where a primitive value will not suffice
- For example, some objects serve as containers of other objects
  - `Vector`..

Wrapper Classes

- Wrapper classes also contain static methods that help manage the associated type
- For example, the `Integer` class contains a method to convert an integer stored in a `String` to an `int` value:

```
num = Integer.parseInt(str);
```

- The wrapper classes often contain useful constants as well
- For example, the `Integer` class contains `MIN_VALUE` and `MAX_VALUE` which hold the smallest and largest `int` values

AUTOBOXING

- *Autoboxing* is the automatic conversion of a primitive value to a corresponding wrapper object:

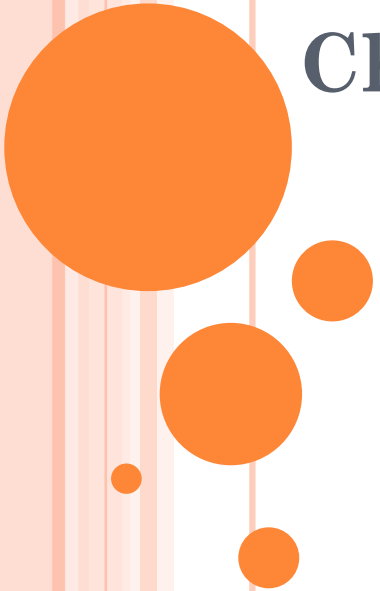
```
Integer obj;
int num = 42;
obj = num;
```

- The assignment creates the appropriate `Integer` object
- The reverse conversion (called *unboxing*) also occurs automatically as needed

Converting Strings to Numbers

```
public class ValueOfDemo {
  public static void main(String[] args) {
     // this program requires two arguments
         if (args.length == 2) {
         //convert strings to numbers
         float a = Float.parseFloat(args[0]);
         float b = Float.parseFloat(args[1]);

         //do some arithmetic
         System.out.println("a + b = " + (a + b) );
         System.out.println("a - b = " + (a - b) );
         System.out.println("a * b = " + (a * b) );
         System.out.println("a / b = " + (a / b) );
         System.out.println("a % b = " + (a % b) );
     } else {
       System.out.println("This program requires two
                           command-line arguments.");
     }
  }
}
```

What's missing?



Chapter 7: File I/O

File I/O - Text Files

- A computer user distinguishes "text" (ASCII) files and "binary" files.
- A **text file** is in reality a binary file (like all files) that contains text (ASCII characters)
- Each line terminates with a "newline" character (or a combination, carriage return plus line feed).
- Examples of text files:
  - Any plain-text file, typically named *something*.txt
  - Source code of programs in any language (e.g., *Something*.java)
  - HTML documents

File I/O - Binary Files

- A "binary" file contains any information, any combination of bytes. Only a programmer / designer knows how to interpret it.
- Different programs may interpret the same file differently (e.g., one program displays an image, another extracts an encrypted message).
- Examples:
  - Compiled programs (e.g., *Something.class*)
  - Image files (e.g., *something.gif*)
  - Music files (e.g., *something.mp3*)
- Any file is (and can be treated as) a binary file (even a text file, if we forget about the special meaning of CR-LF, i.e. end of line).

File I/O – Streams of bytes

- A **stream** is generally a sequence of data
- Java byte streams are used to perform input and output of 8-bit bytes

- Two kinds of streams:
  - **Connection** (low-level – only work with bytes)
  - **Chain** (can do numbers, strings and other Objects)

- And another distinction:
  - **InputStream:** used to read data from a source.
  - **OutputStream:** used to write data to a destination.

- Most common connection stream classes are:
  - **FileInputStream**
  - **FileOutputStream**

File I/O – Streams of characters

- Java character streams are used to perform input and output of 8-bit characters

- Generic classes here:
  - **InputStreamReader**
  - **OutputStreamWriter**

- Most common connection stream classes are:
  - **FileReader**
  - **FileWriter**

File I/O – formatted text

- **`Scanner`** class. (Or `BufferedReader`)

- `nextInt(), nextDouble(), nextLine()` etc
  - *Example*: int n = inStream.nextInt();
- To test for end of input use methods: `hasNextInt(), hasNextLine()` etc

- **`PrintWriter`** class (Or `BufferedWriter`)
- `print(), println()`
- `close()`

READING TEXT - SCANNER

- You can use the methods **nextInt()**, **nextDouble()**, **nextLine()** etc to read from the named files.
  - ***Example*: int n = inStream.nextInt();**

- To test for end of input use methods: **hasNextInt()**, **hasNextLine()**

Writing text - PrintWriter

- The class **PrintWriter** is the preferred stream class for writing to a text file.
- Similar to Scanner for reading

```
PrintWriter printWriter = new PrintWriter(new
                    FileOutputStream("diary.dat"));
```

- Like Scanner, constructor needs a FileOutputStream associated with a file
  - Or a FileWriter or a File
- Use printWriter.println(String …) to write

File I/O – Most Common Classes

	Byte-based		Character-based	
Basic	`InputStream`	`OutputStream`	`InputStream Reader`	`OutputStream Writer`
Files	**`FileInputStream`**	**`FileOutputStream`**	**`FileReader`**	**`FileWriter`**
Buffered	`BufferedInputStream`	`BufferedOutputStream`	`BufferedReader`	`BufferedWriter`
Formatted text			**`Scanner`**	**`PrintWriter`**

READING FROM A FILE (BY CHARACTER) - 1

```
import java.io.*;
public class test2 {
public static void main(String [] args) throws Exception{
  FileReader inone = new FileReader(“myfile.txt");
  int t=inone.read();
  while (t!=-1)  {
        System.out.print((char)t);
        t=inone.read();
        }
  }
}
```

Reading from a file (by character) - 2

- Reading from file ‘myfile.txt’ a single character at a time.

- The method **read()** returns an int: either
  - the *Unicode value of the character just read in, or*
  - *-1 if* we have reached the end of file.

- We must cast the value just read in as a character in order to print it out properly.

Reading from a File (by line)

```
import java.io.*;
import java.util.Scanner;
public class test1  {
public static void main(String[] args) throws Exception {
  Scanner in =new Scanner(new FileInputStream(“myfile.txt"));
   while (in.hasNextLine()) {
       System.out.println(in.nextLine()); //reads String
        }
    }
  }
```

File Read/Write by byte

```
public class CopyFile {
  public static void main(String args[]) throws IOException{
       FileInputStream in = new FileInputStream("input.txt");
       FileOutputStream  out = new FileOutputStream("output.txt");
       int c;
       while ((c = in.read()) != -1) {
              out.write(c);
       }
       in.close();
       out.close();
  }
}
```

File Read/Write by character

```
import java.io.*;
public class copy {
  public static void main(String[] args) throws Exception {
       FileReader inone = new FileReader("file1.dat");
       FileWriter outone =new FileWriter("file2.dat");
       int t=inone.read();
       while (t!=-1) {
              outone.write(t);
              t=inone.read();
       }
       outone.close();
  }
}
```

File Read/Write by line

```
import java.io.*;
public class Copy {
  public static void main(String[] args) throws Exception {
        Scanner scanner = new Scanner(new
                              FileInputStream("file1.dat"));
        PrintWriter pWritter = new PrintWriter (new
                              FileOutputStream("file2.dat"));

        while (scanner.hasNextLine()) {
               pWritter.println(scanner.nextLine());
        }
        scanner.close();
        pWritter.close();
  }
}
```

EXERCISE 7.1

Write a class CopyFileInCapitals that takes two filenames as command line arguments, as for example:

```
java  CopyFileInCapitals  input.txt  output.txt
```

The program should copy all contents of input.txt to output.txt (erasing its previous contents, if any), converting all lower case letters into capital letters.

EXERCISE 7.1

```
import java.io.*;
public class Copy {
  public static void main(String[] args) throws Exception {
       Scanner scanner = new Scanner(new
                                    FileInputStream(args[0]));
       PrintWriter pWritter = new PrintWriter (new
                     FileOutputStream(args[1]));

       while (scanner.hasNextLine()) {
          pWritter.println(scanner.nextLine().toUpperCase());
       }
       scanner.close();
       pWritter.close();
  }
}
```

EXERCISE 7.1

```
public static void main(String[] args) throws IOException {
            FileReader fr = null;
            FileWriter fw = null;

            try {
               fr = new FileReader("input.txt");
               fw = new FileWriter("output.txt");

               int c;
               while ((c = fr.read()) != -1) {

               String c2 =
                 Character.toString((char) c).toUpperCase();
```

EXERCISE 7.1

```
                fw.write(c2);
              }
            } finally {
              if (fr != null) fr.close();
              if (fw != null) fw.close();
            }
}
```

EXERCISE 7.2

Write a method named concatenateFiles that reads two existing text files, named input1.txt and input2.txt, and concatenates them into a new file named output.txt. Concatenation should occur by appending the contents of input2.txt after the contents of input1.txt into output.txt. For example if input1.txt contains the following lines:

```
Line1
Line2
Line3
```

and input2.txt contains

```
LineA
LineB
LineC
```

then output.txt should be

```
Line1
Line2
Line3
LineA
LineB
LineC
```

EXERCISE 7.2

Answer

```
public void concatenateFiles() throws IOException{
      FileReader in1, in2;
      FileWriter out;
      in1 = new FileReader ("input1.txt");
      in2 = new FileReader ("input2.txt");
      out = new FileWriter ("output.txt");
      int c;
      while((c=in1.read())!=-1){
            out.write(c);
      }
      while((c=in2.read())!=-1){
            out.write(c);
      }
}
```

EXERCISE 7.2

Alternative answer with BufferedReader and BufferedWriter

```
public void concatenateFiles() throws IOException{
   BufferedReader in1, in2;
   BufferedWriter out;
   in1 = new BufferedReader (new FileReader ("input1.txt"));
   in2 = new BufferedReader (new FileReader ("input2.txt"));
   out = new BufferedWriter (new FileWriter ("output.txt"));
   String line;
   while((line=in1.readLine())!=null){
        out.write(line);
   }
   while((line=in2.readLine())!=null){
        out.write(line);
   }
 }
```

EXERCISE 7.3

Describe what the following code does.

```
import java.io.*;
import java.util.Scanner;
public class A
{
        public static void main(String[] args) throws Exception
        {
                Scanner in =new Scanner(new
                                FileInputStream("file1"));
                while (in.hasNext ())
                {
                        String t= in.nextLine();
                        if (t.startsWith(args[0])) {
                                System.out.println(t);
                        }
                }
        }
}
```

Answer: Opens a file named file1 and looks for strings starting with the String passed as the first argument to the program

EXERCISE 7.4

Write a method `static void copy(String g1, String g2)` *which copies the contents of the text file whose name is given by the parameter* `g1` *to the file whose name is given by the parameter* `g2`.

EXERCISE 7.4

```
static void copy (String g1, String g2){
        try{
                Scanner in =new Scanner(new FileInputStream(g1));
                PrintWriter out =new PrintWriter(new
                                FileOutputStream(g2));
                while (in.hasNextLine()){
                        out.println(in.nextLine());
                }
                in.close();
                out.close();
        }
        catch(IOException e){
                System.out.println("trouble");
                return;
        }
}
```

EXERCISE 7.5

Consider the following program

```
import java.io.*;
public class Q5a{
  public static void doit(String s) throws Exception{
        BufferedReader inone=new BufferedReader(new FileReader(s));
        int t=inone.read();
        while (t!=-1){
                System.out.print((char)t);
                t=inone.read();
        }
  }
  public static void main(String[] args) throws Exception{
        doit(args[0]);
  }
}
```

EXERCISE 7.5

(i) What will be the output if we run Q5a class with Q5a.java as the input parameter?

(ii) What does it mean when the variable t gets the value -1?

(iii) What would happen if we ran Q5a with an empty input file? Why?

Ans: (i) The contents of Q5a.java will be printed on screen

(ii) This denotes that the end of the file has been reached

(iii) the condition t=-1 would be false, so the code would never enter the while loop and nothing would be printed. Notice that the program will run, as the file does exist, though it is empty.

EXERCISE 7.6

Write a complete Java program called `writeEveryOther` *that writes the contents of the file to another file, omitting every second character. Both file names should be passed to the program as command line arguments. You may assume that method* `everyOther` *from the previous part has been correctly implemented.*

```
public class WriteEveryOther {
//public String everyOther(String s){…
  void copy(String g1, String g2){
    try{
      BufferedReader inone =new BufferedReader(new FileReader(g1));
      BufferedWriter outone =new BufferedWriter(new FileWriter(g2));
      String line;
      while((line=inone.readLine())!=null){
        outone.write(everyOther(line));
      }
    }
    catch(IOException e){
      System.out.println(g1 + " does not exist");
      return;
    }
  }
```

EXERCISE 7.6

(continued)

```
        public static void main(String[] args) throws Exception{
                WriteEveryOther weo = new WriteEveryOther();
                weo.copy(args[0], args[1]);
        }
}
```

Mock exam 2019: Question 5b

Briefly describe the functionality of the following program.

```
public static void main(String[] args) {
    try {
        Scanner scanner = new Scanner(new FileReader(args[0]));
        while (scanner.hasNextLine()) {
            String line = scanner.nextLine();
            String[] lineArray = line.split(" ");
            for(int i=0;i<lineArray.length;i++) {
             System.out.println(lineArray[i]);
            }
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

Answer: Reads a text file, passed as a command line argument, splits the words and then prints them one by one in separate lines

Mock exam 2019: Question 5c

Write a method named

`replace(String input, String seed, String replacement)`

that replaces all occurrences of `seed` *within* `input` *with* `replacement` *and prints the new string. The replacement is case sensitive. You may use any String API method* ***except*** *the readily available* `String.replace()` *method.*

For example, the following call of your method with:

"The UK should not be written as uk", "UK", "United Kingdom");

should print

"The United Kingdom should not be written as uk".

Mock exam 2019: Question 5c

- Answer

```
public static String replace2(String input, String
                              seed, String replacement){
        String[] s= input.split (seed);
        String out = s[0];
        for(int i=1; i<s.length; i++){
                out+=(replacement+s[i]);
        }
        return out;
}
```

```
//though this wouldn't detect the seed pattern if it
was at the beginning or at the end of the phrase
```

Mock exam 2019: Question 5c

- Answer

```
public static String replace(String input, String seed,
                                    String replacement){
        String[] s= input.split(" ");
        String out = s[0];
        for(int i=1; i<s.length; i++){
                if(s[i].equals(seed)) {
                        s[i]=replacement;
                }
                out+=" "+s[i];
        }
        //System.out.println(out);
        return out;
}
//though this wouldn't detect the seed pattern if it
 was not separated with whitespace from the rest.
```

Mock exam 2019: Question 5c

- Attempt:

```
String[] sArray= input.split(" ");;
for(String s1 : sArray)
      if(s1.compareTo(seed) == 0){
            s1=replacement;
      }
}
```

This doesn't work because changing the value of a local variable will never change anything else - it *just* changes the local variable. That doesn't change the contents of the array. The iteration variable is just a *copy* of the array element

Mock exam 2019: Question 5d

Write a method with signature

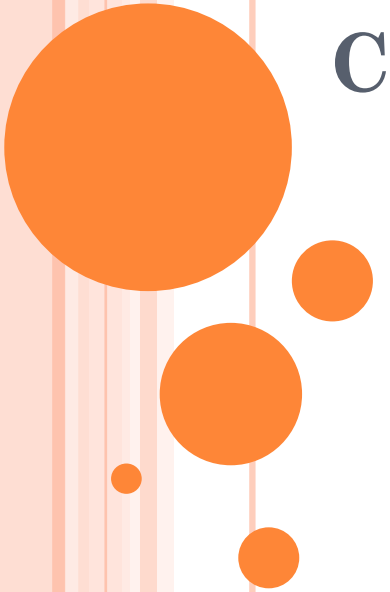
```
void copy(String f1, String f2, String
seed, String replacement)
```

The method reads a file named f1 and copies its contents to another file f2, after having replaced all occurrences of seed with replacement. You can use the method replaceString from the previous part and assume that it works correctly.

Mock exam 2019: Question 5d

- Answer

```
public static void copy(String f1, String f2, String
seed, String replacement){
  try {
    Scanner scanner = new Scanner(new FileReader(f1));
    PrintWriter pw = new PrintWriter(new FileWriter(f2));
    while (scanner.hasNextLine()) {
      String line = scanner.nextLine();
      pw.println(replace(line, seed, replacement));
    }
    scanner.close();
    pw.close();
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```



Chapter 8: Searching in Arrays

Linear Search

- Linear Search - Scans the array comparing the target value to each element:

```
public int sequentialSearch (Object arr [ ], Object value )
{
    for (i = 0;  i < arr.length ;  i++)
    {
        if  (value.equals (a [ i ]))
            return i;
    }
    return -1;
}
```

For primitive data types it is value == a [i]

Linear Search

- The average number of comparisons (assuming the target value is equal to one of the elements of the array, randomly chosen) is about $n / 2$ (where n = *arr.length*).
- Worst case: n comparisons
- Also n comparisons are needed to establish that the target is not in the array.
- We say that linear search is of order n, i.e. **O(*n*)**

More precisely:
$(1 + 2 + \ldots + n) /$
$= (n + 1) / 2$

Finding Max and Min

- Find the max value in the array

```
public double findMax(double arr [ ])
{
    double max = arr [ 0 ];

    for ( i = 1;  i < arr.length;  i++)
    {
        if  (a [ i ] > max)
            max = a [ i ];
    }

    return max;
}
```

Finding Max and Min

- Find the **position** of max value in the array

```
public int findMaxPos(double arr [ ])
{
    maxPos =  0 ;

    for ( i = 1;  i < arr.length;  i++)
    {
        if  ( a [ i ] > a [ maxPos ] )
            maxPos = i ;
    }

    return i;
}
```

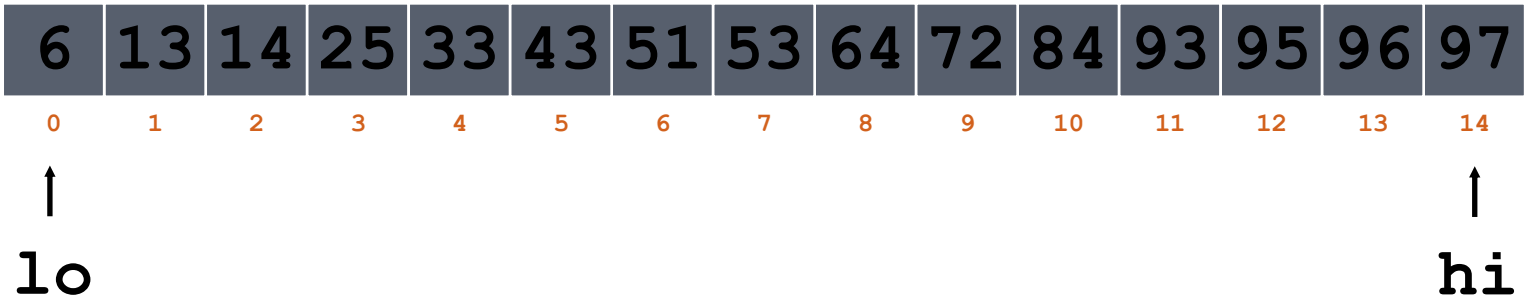
Binary Search

- For numbers pre-arranged in ascending (descending) order
  - or any type of "ordinal" objects.
- A "divide and conquer" algorithm:

1. compare the target value to the middle element;
2. if equal, all set
3. if smaller, apply binary search to the left half;
4. if larger, apply binary search to the right half.

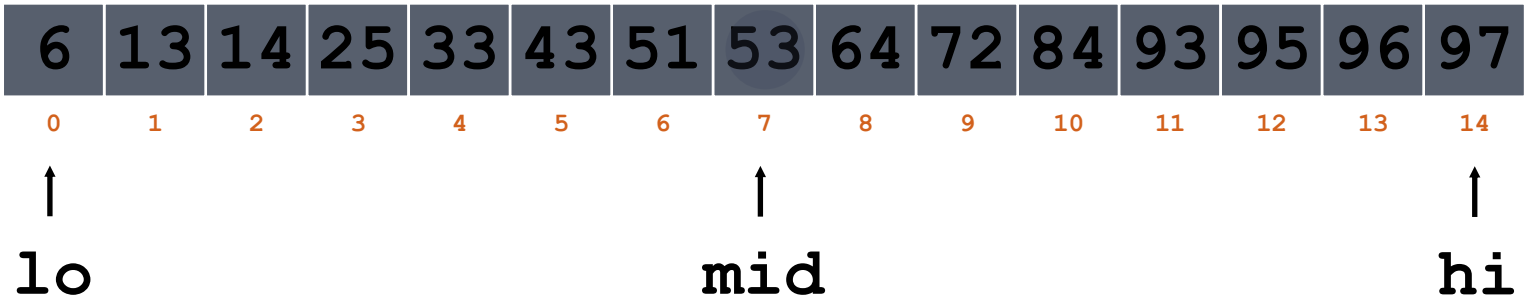
Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.



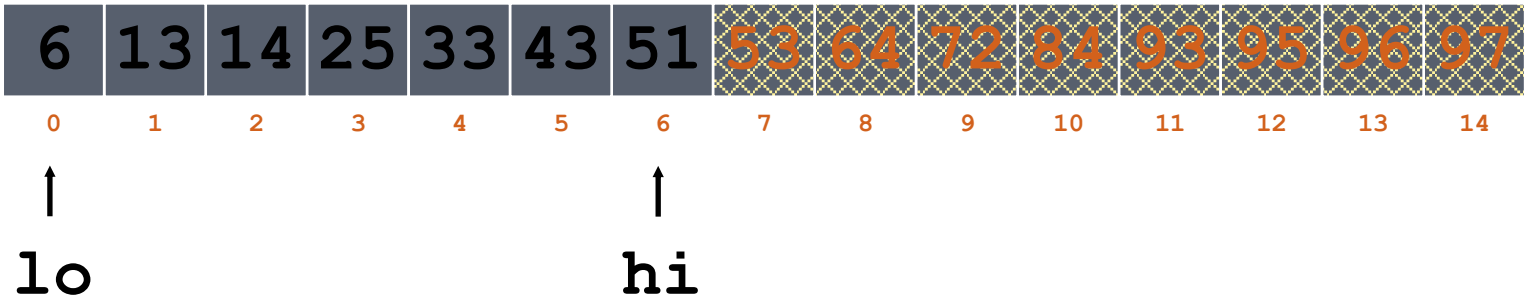
Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.



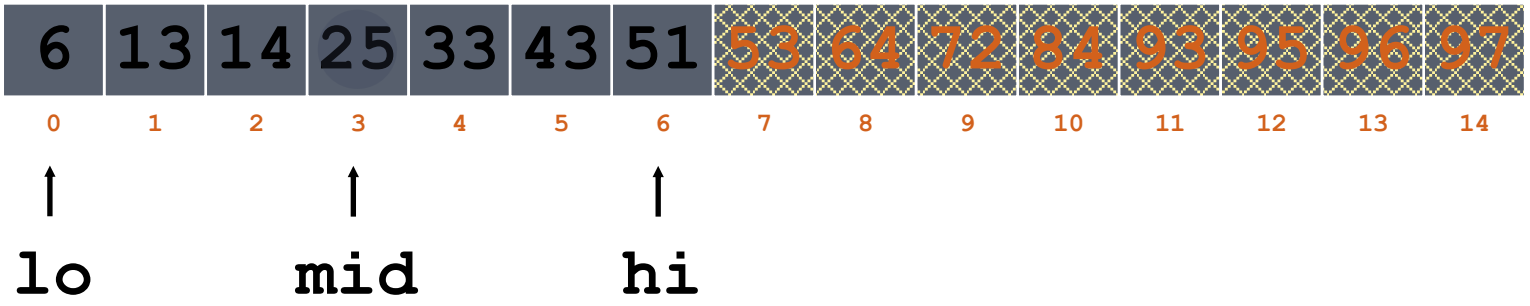
Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.



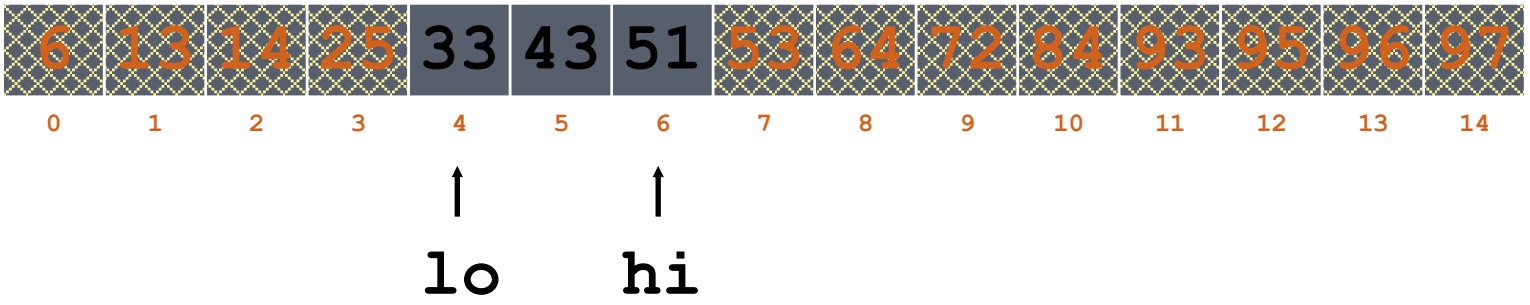
Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.



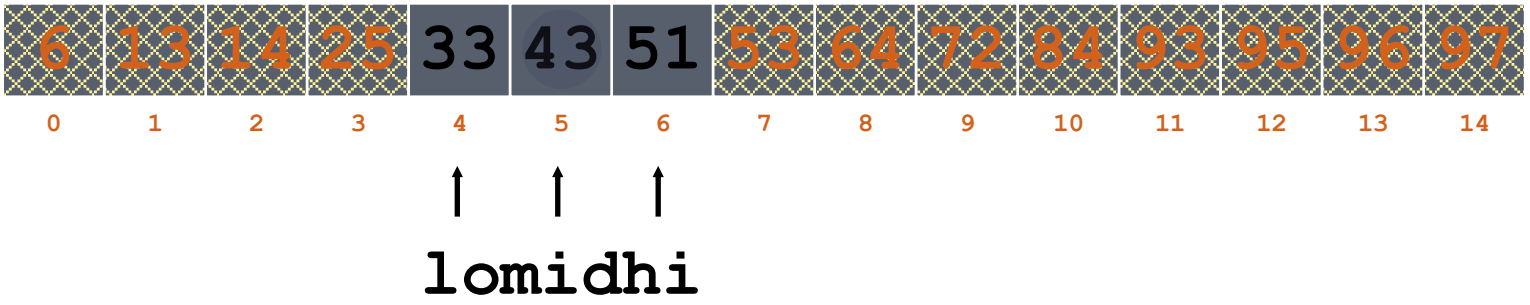
Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.



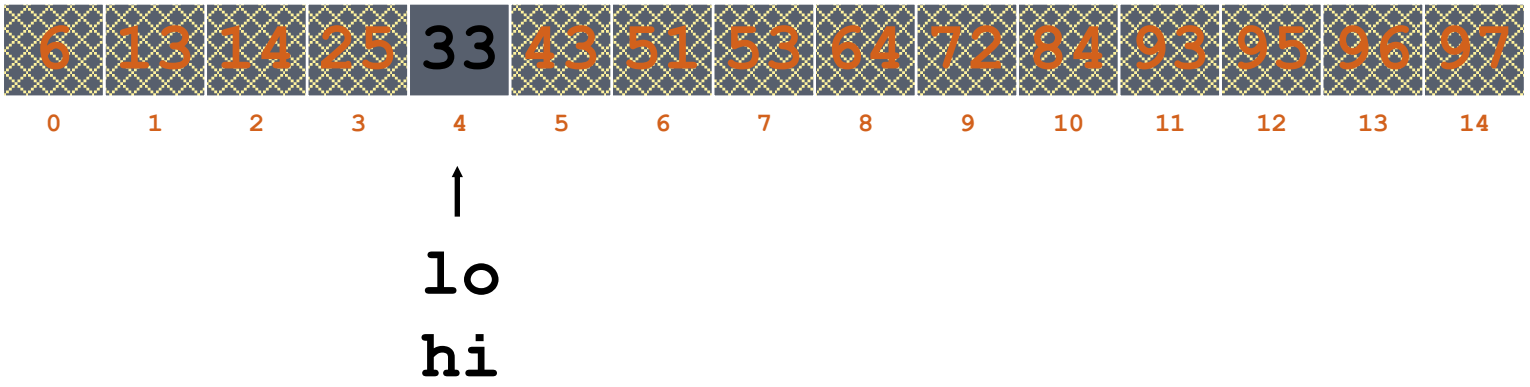
Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.



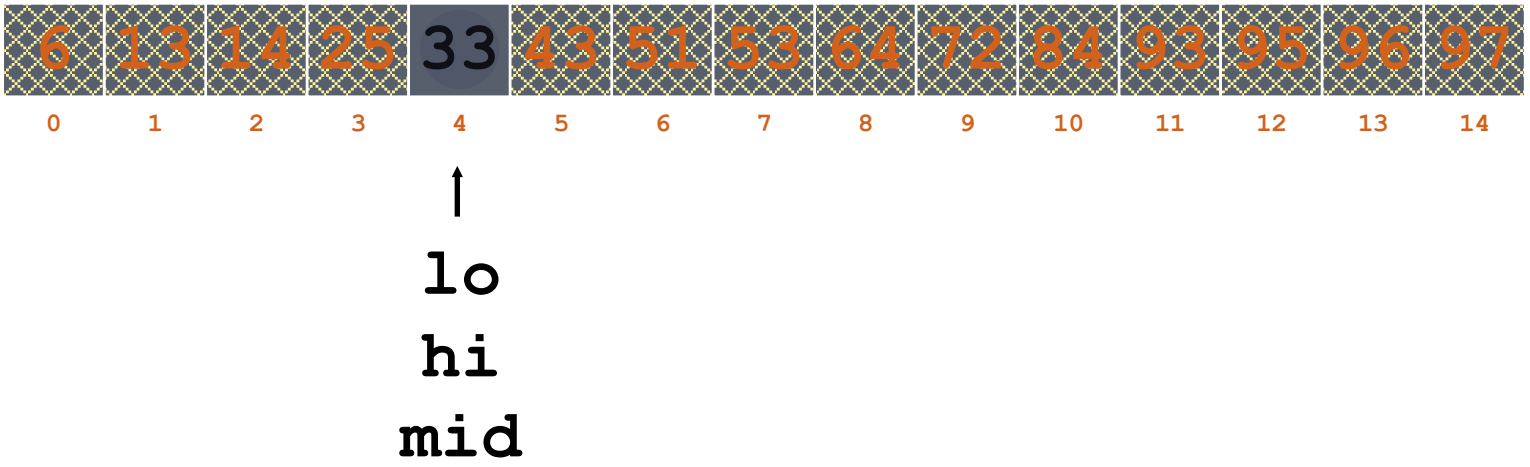
Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.



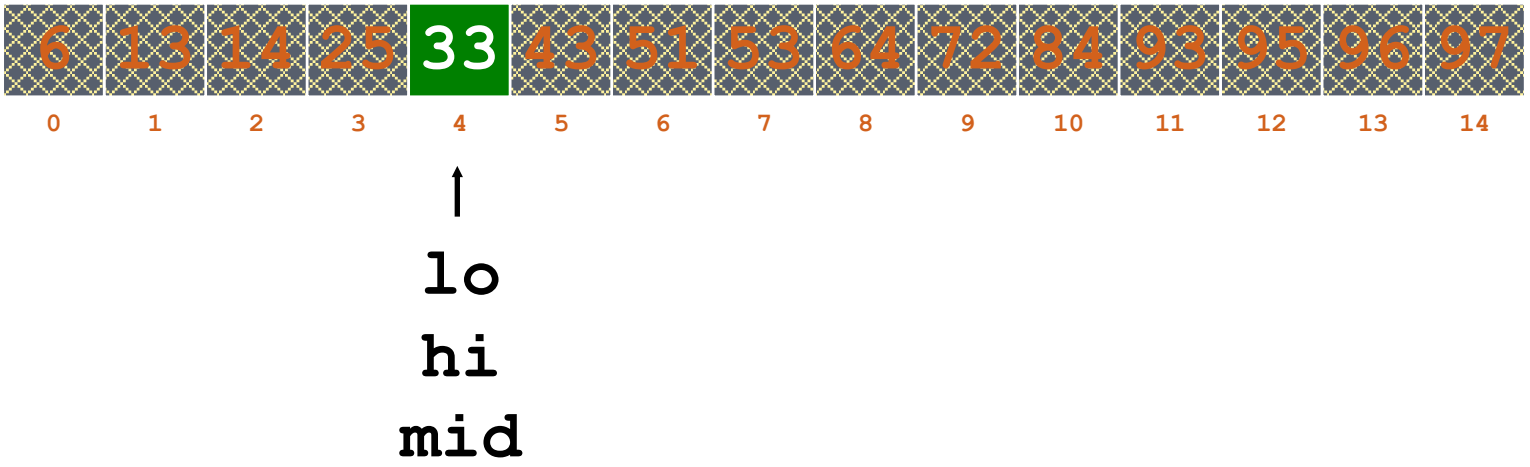
Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.



Binary Search

- Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.
- Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.
- Ex. Binary search for 33.



Binary Search

- Iterative implementation:

```
public int binarySearch (int a[ ], int value, int lo, int hi)
{
   while (lo<= hi)
   {
       int mid= (lo+ hi) / 2;

       if ( value == a[mid] )
          return mid;
       else if ( value < a[mid] )
          hi= mid– 1;
       else                          // if ( value > a[mid] )
          lo= mid+ 1;
    }
    return -1;                       // Not found
}
```

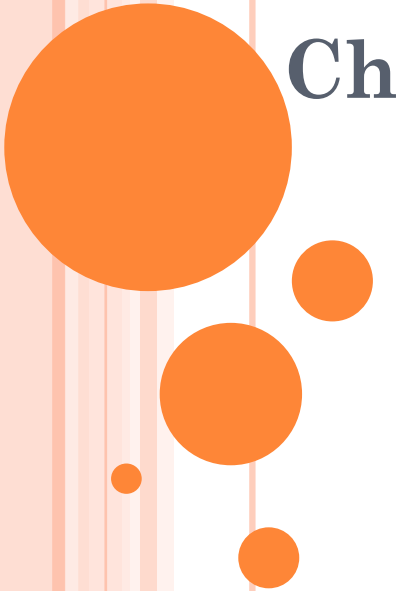
Binary Search

- Recursive implementation:

```
public int binarySearch (int a [ ], int value, int lo, int hi)
{
   int mid= (lo + hi) / 2;

   if (value == a[mid] )
      return mid;

   else if ( value < a[mid] && lo< mid)
      return binarySearch (a, value, lo, mid– 1);

   else if ( value > a[mid]  &&  mid< hi)
      return binarySearch (a, value, mid+ 1, hi);

   else
      return -1;  // Not found
}
```

Binary Search

- The number of times a number ***n*** can be cut in half and not go below 1 $\log_2 n$
- For example, if n is 16, then we can do four divisions by 2. **i.e. 16/2 = 8 ; 8/2 = 4; 4/2 = 2; 2/2 = 1**
- So $\log_2 16 = 4$
- Binary search is **O ($\log_2$n)**. A very fast method: only 20 comparisons are needed for an array of 1,000,000 elements



Chapter 9: Exceptions

Things that could go wrong

- Database call:
  - connection lost
- Get a text file:
  - not there
- Parse a number from a String:
  - not in correct format
- …

- How can we handle these?

Exceptions

- An *exception* is an event, which occurs during the execution of a program, that disrupts its normal flow
- When an error occurs inside a method, the method creates an object and hands it off to the runtime system.
- This object, called an *Exception object*, contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called *throwing an Exception*.

Handling exceptions

- try
  - Identifies a block of code where an exception might be thrown
- catch
  - Process/handle named type of exception
- finally
  - Code block that will always be executed

Exception handling example

```
try {
    statement(s)
}
catch( SubClassException sce ) {
    statement(s)
}
catch( Exception ex )  {
    statement(s)
}
finally {
    statement(s)
}
```

Catch Blocks

- Handles the exception (according to exception type)
- Responsible for executing adequate code when something goes wrong

Finally

- A block of code executed no matter what

- Things for finally:
  - Release resources (cleanup)
    - File handles
    - Database connections and statements
  - Gather statistics (elapsed time for method execution)
  - …

Where to handle problems?

- Catch and handle where appropriate

- If you don't want to or can't handle it you can allow the exception to pass up the method call chain – declare that your method will "throw" the exception types that you can't handle

Throwing out of a method

- throws exceptionList
  - If your method throws any checked exceptions, your method declaration must indicate the type of those exceptions.

- Example:

```
public void myMethod() throws MyException,
  MyOtherException {
     if( bigProblem ){
         throw new MyException();
     }
}
```

Exceptions

- Checked:
  - they require being caught and handled during compile time
  - declared by the programmer to describe an error condition

- Unchecked:
  - not checked/verified during compile time
  - less predictable, depending on data
  - (e.g. array out of bounds, division by zero etc)

Exception examples

- Checked:
  - `FileNotFoundException`,
  - `ClassNotFoundException`,
  - `IllegalAccessException`,
  - `NoSuchFieldException`,
  - `NoSuchMethodException`,
- Unchecked:
  - `ArithmeticException`,
  - `ArrayIndexOutOfBoundsException`,
  - `IllegalArgumentException`,
  - `NullPointerException`

EXERCISE 9.1

(i) What is the output of the following program?

```
public class NumberHandler {
    public static void main(String []args){
        String s = "one";
        try{
            int x = Integer.parseInt(s);
            System.out.println("Output is: "+x);
         } catch (Exception e){
            System.out.println("Failed...");
         } finally {
            System.out.println("Input was: "+s);
         }
    }
}
```

- Ans: Failed...
  Input was: one

EXERCISE 9.2

What is the output of the execution of the main method in the following code?

```
class MyException extends Exception{
    MyException (){
        super ("My Exception");
    }
}

class YourException extends Exception {
    YourException (){
        super ("Your Exception");
    }
}
```

EXERCISE 9.2

```
class ChainDemo {
   public static void main (String [] args){
      try{
         someMethod1 ();
      }
      catch (MyException e){
            System.out.println (e.getMessage ());
      }
   }
   static void someMethod1 () throws MyException {
      try {
         someMethod2 ();
      } catch (YourException e) {
         System.out.println (e.getMessage ());
         MyException e2 = new MyException ();
           throw e2;
      }
   }
   static void someMethod2 () throws YourException{
      throw new YourException ();
   }
}
```

EXERCISE 9.2

Ans:

```
Your exception
My Exception
```

EXERCISE 9.3

What is the output of the following program?

```
class OneTwo{
	public static void main(String[] args){
		try{
			Integer.parseInt("one");
			System.out.println("one");
		}
		catch(Exception e){
			System.out.println("two");
		}
	}
}
```

Answer: two

EXERCISE 9.5

For each of the following statements say whether it is true or false.

(i) *All try/catch blocks must have a finally block.*
(ii) *A try/catch block is limited to two or less catch blocks.*
(iii) *The finally block will always be executed.*
Ans:

(i) F

(ii) F

(iii) T

EXERCISE 9.7

Using try and catch and the method Integer.parseInt, write a complete Java program called ProvideInt. The program prompts the user to provide an integer value. If the input is not valid, the program repeatedly asks the user to re-enter one.

```
public class ProvideInt {
	public static boolean isInt(String s){
		try{
			Integer.parseInt(s); return true;
		}
		catch(Exception e){
			return false;
		}
	}
```

EXERCISE 9.7

continued

```
public static void main(String[] args){
                System.out.println("Please provide an integer");
                Scanner scan = new Scanner(System.in);
                String input = scan.nextLine();
                while(!isInt(input)){
                        System.out.println("Invalid input. Please
try again");
                        input = scan.nextLine();
                }
                System.out.println("Thanks for your input:
"+input);
        }
}
```

Mock exam 2019: Question 1c

Write a complete Java program that prints out the product of all command line arguments supplied to it. You can assume that all of them are integers.

Answer

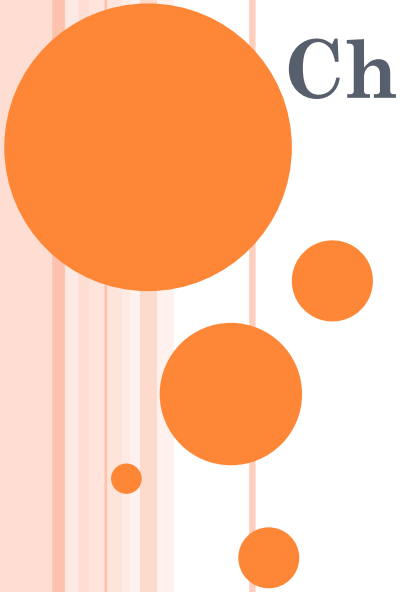
```
public class ProdOfInputs {
        public static void main(String[] args)
        {
            int total=1;
            for (int i=0;i<args.length;i++)
               total*= Integer.parseInt(args[i]);
            System.out.println(total);
        }
}
```

Mock exam 2019: Question 3c

Using try and catch and the method Integer.parseInt, write a static method, isInt that takes a String as a parameter and returns true if the string contains only digits and false otherwise.

Answer:

```
public class IntException {
	public static boolean isInt(String s){
		try{
			Integer.parseInt(s);
			return true;
		}
		catch(Exception e){
			return false;
		}
	}
}
```



Chapter 10: Static and Final

METHODS – 'REGULAR' VS. STATIC

- A regular method requires an instance – an object
  - 'Instance method'

- A static method can be called without an instance of the method's class being on the heap
  - 'Class method'

- **`Math.random()`** vs **`myClass.go()`**

Static Method
No objects

Non-Static Method

requires an instance - object

Static variables

- Shared by all members of a class
- Only one copy per class (… and not per instance)
- Non-static methods can use static methods/variables but not the other way around.

Final

- A variable can be marked as **static** AND **final**
  - Similar to constants
- For security
- If a class is final – its methods are final
  - cannot be extended
- Final must be initialized in declaration or constructor.
- Final can be used on instance variables and methods – compiler will block any attempt to change or override.

EXERCISE 10.1

Can you identify two problems with the following class?

```
public class YY
{
        public static void main(String[] args)
        {
                int x=0;
                final int y = 1;
                y=x;
                x="22";
                System.out.println(x);
                System.out.println(x);
        }
}
```

Ans:

y=x -->(y final)

x='s' --> x is an int

EXERCISE 10.2

For each of the following statements, say whether it is true or false

(i) *Static methods cannot be overloaded.*
(ii) *Static methods can be overridden.*
(iii) *Constructors can be static.*
(iv) *Constructors can be overloaded but not overridden.*
(v) *Static variables represent global state.*

Ans:

(i) F - you can overload a static method.

(ii) F - you cannot override a static method

(iii) F - it makes no sense for a constructor to be **static**.

(iv) T - Constructors can be overloaded. Having a constructor in a Child class with the name of the parent class does not make sense.

(v) T - Used along all objects with the Class' name.

EXERCISE 10.3

The following classes will not compile. Why?

```
public class QQ{
        double z;
        public static void main (String[] args){
                z=Double.parseDouble(0.0);
        }
}
```

Ans:

ParseDouble accepts argument of type String

Also static method cannot call non-static variable

EXERCISE 10.4

The following classes will not compile. Why?

```
public class XX {
	public static void main(String[] args){
		int i = 1;
		doTheMath(i);
	}
	public void doTheMath(int n){
		for(int i=0; i<n; i=i+2){
			System.out.println(i+2);
		}
	}
}
```

Ans: A static method cannot call an instance one directly.

Mock exam 2019: Question 3b

For each of the following classes, say whether they will produce compilation errors and, if they do, briefly explain the error.

(i)

```
public class Q3bi {
	final static int z = 1;
	public static void main(String[] args) {
		for (int i = 1; i< 100; i++)
			z++;
		System.out.println(z);
	}
}
```

Ans: `Final field z cannot be assigned`

Mock exam 2019: Question 3b

For each of the following classes, say whether they will produce compilation errors and, if they do, briefly explain the error.

(ii)

```
class Q3bii {
	final static int k = 11;
	public static void main(String[] argsss) {
		if(argsss.length>1)
			System.out.println(argsss[0]+k);
	}
}
```

Ans: `No error`

Mock exam 2019: Question 3b

For each of the following classes, say whether they will produce compilation errors and, if they do, briefly explain the error.

(iii)

```
class Q3biii{
	final int k = 11;
	public static void main(String[] args) {
		System.out.println(k);
	}
}
```

Ans: Non-static field cannot be accessed from static method