CS151 Intro to Data Structures

Java Basics

Administrivia

- Course website
  - `BMC-CS-151.github.io`
  - Assignments and lab instructions, syllabus
- ~apoliak/handouts/cs151
  - labs
  - sample code handouts
  - data sets
  - lecture notes
- Midterm: Wednesday 25th (Wednesday after Fall break)

Administrivia

- Piazza:
  - Asynchronous communication
  - Can post anonymously (anonymous just to classmates)
  - Answer your peers questions!
    - Counts for participation grade

- Gradescope:
  - Submit all assignments
  - Can request re-grade requests

Labs and TAs

- Completed labs must be checked off by TAs, either in lab or during office hours
- Demo code that does what the marked exercises ask
- Completed labs will be a portion of your grade
- TA hours start - TBD

First Things

- CS server account
  - Make sure you can log in
  - Email David Diaz if encountering issues (ddiaz1@brynmawr.edu)

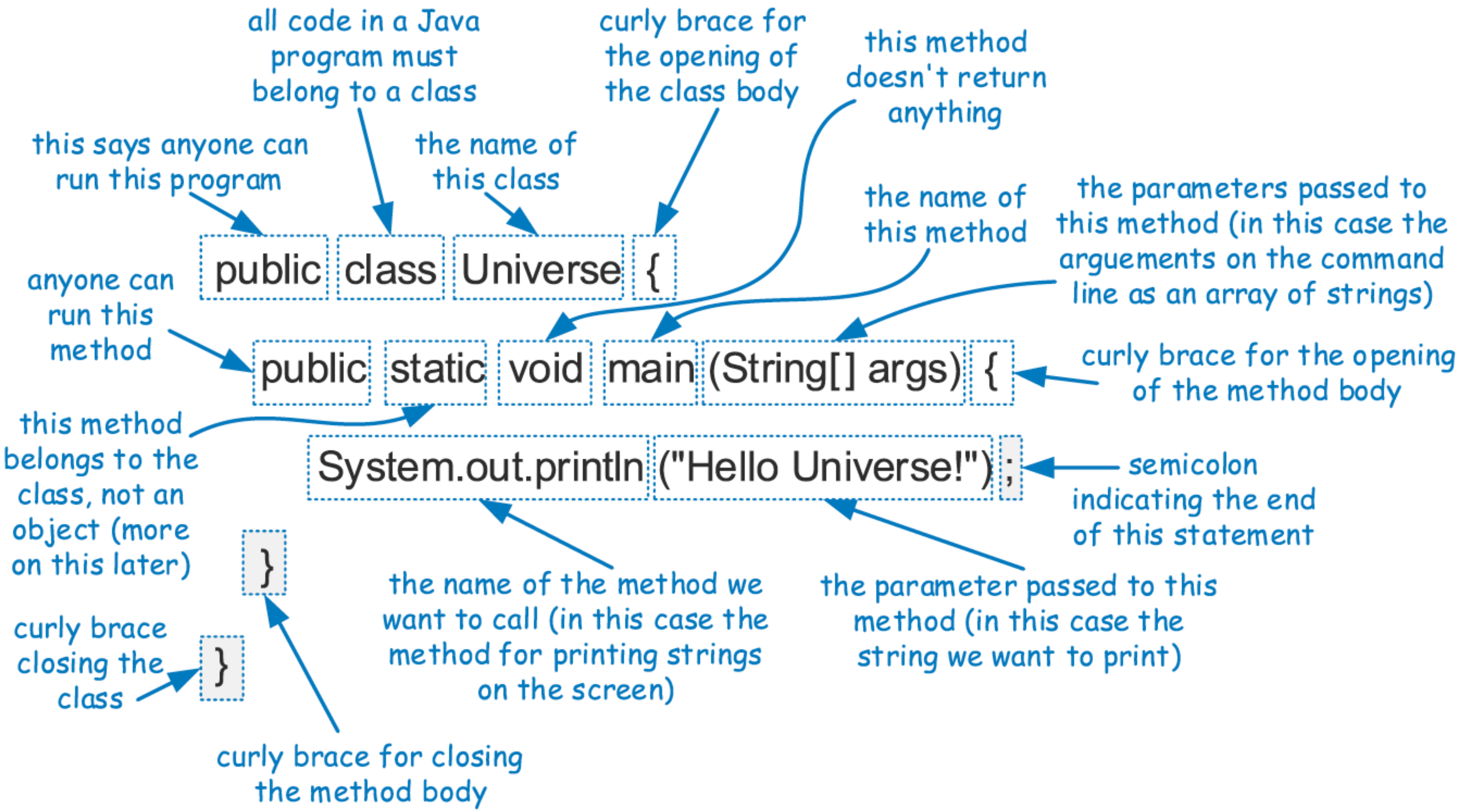
- Lab: Park 231/W 2:40pm-4:00pm
- Lab00: ideally completed already, getting up and running with vim and linux

- Lab attendance is required. Lab exercise must be completed BEFORE you start your assignments
- Software: vim, Java, or just ssh

Outline

- Data Types
- Objects
- String review
- Input (Scanner)
- OOP (Inheritance)
- File I/O, Exceptions
- ***Not reviewing:***
    - Methods
    - Loops

An Example Program



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

this says anyone can run this program

all code in a Java program must belong to a class

the name of this class

curly brace for the opening of the class body

this method doesn't return anything

the name of this method

the parameters passed to this method (in this case the arguements on the command line as an array of strings)

anyone can run this method

curly brace for the opening of the method body

this method belongs to the class, not an object (more on this later)

semicolon indicating the end of this statement

the name of the method we want to call (in this case the method for printing strings on the screen)

the parameter passed to this method (in this case the string we want to print)

curly brace closing the class

curly brace for closing the method body

Java: A compiled language

- Java program in .java (source code)

- Compiler create .class file (byte code)

- Java Virtual Machine (JVM) execute the code

Java Basics

- Name of main class and file must agree
  - `class Driver` <--> `Driver.java`
- Compilation
  - `javac Driver.java`
- Execution
  - `java Driver`

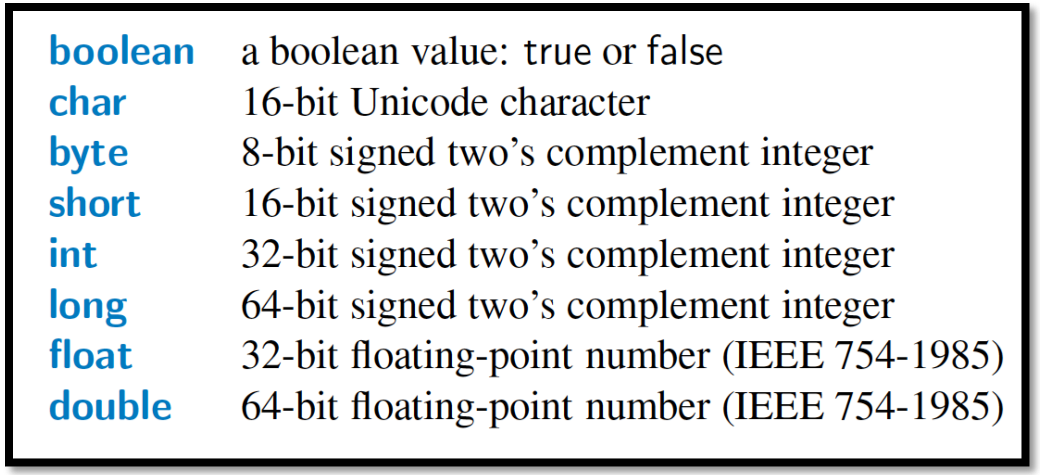
Components of a Java Program

- Statements are placed in *methods*, that belong to class definitions.
- The static method named `main` is the first method to be executed when running a Java program.
- Any set of statements between the braces `{` and `}` define a program block.

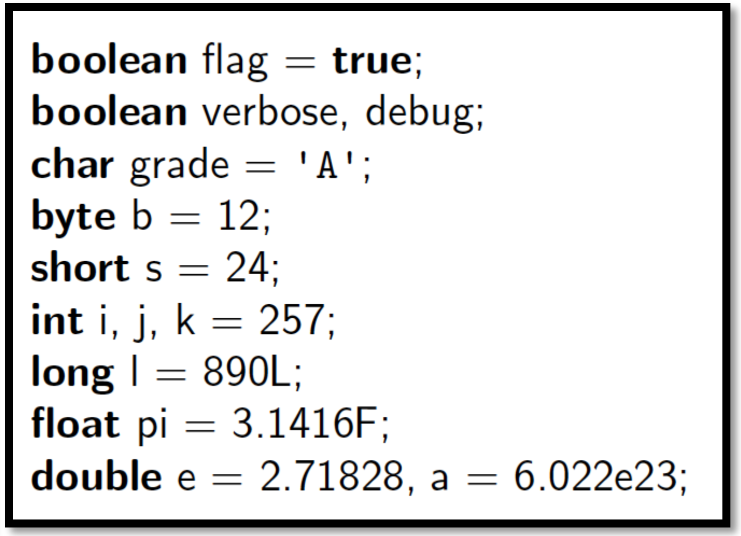
Base/Primitive Types

- Variables must have types
  - base type
- Types define memory used to store the data

- Primitives:



boolean	a boolean value: true or false
char	16-bit Unicode character
byte	8-bit signed two's complement integer
short	16-bit signed two's complement integer
int	32-bit signed two's complement integer
long	64-bit signed two's complement integer
float	32-bit floating-point number (IEEE 754-1985)
double	64-bit floating-point number (IEEE 754-1985)



```
boolean flag = true;
boolean verbose, debug;
char grade = 'A';
byte b = 12;
short s = 24;
int i, j, k = 257;
long l = 890L;
float pi = 3.1416F;
double e = 2.71828, a = 6.022e23;
```

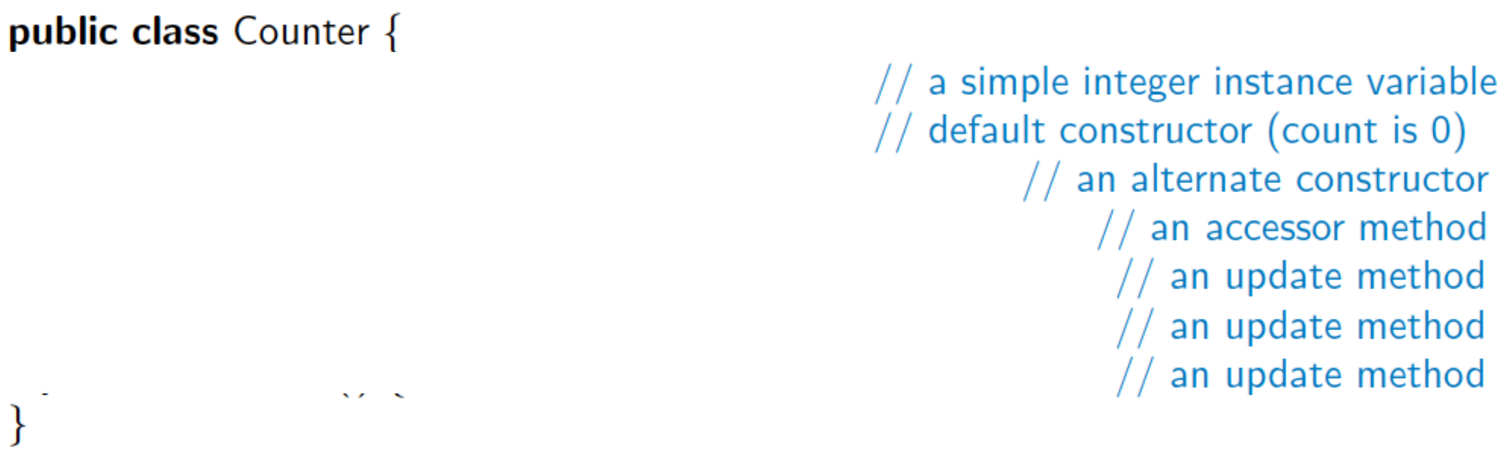
Classes and Objects

- ***Classes*** are blueprints, ***objects*** are instance of the classes

- A class defines:
  - instance variables – what the object stores
  - Methods – how the object functions

- Every variable is either a primitive or a reference to an object
  - Remember those stack frame diagrams!

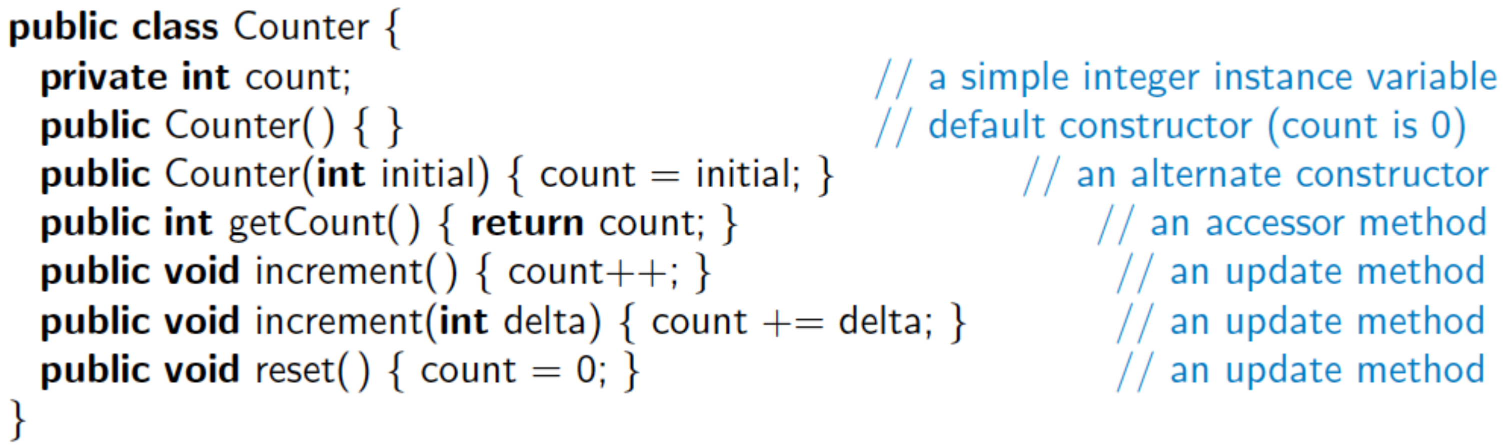
Class Example



```
public class Counter {
                                        // a simple integer instance variable
                                        // default constructor (count is 0)
                                                    // an alternate constructor
                                                         // an accessor method
                                                          // an update method
                                                          // an update method
                                                          // an update method
}
```

- instance variable
- methods
  - constructor
  - accessor
  - update

Class Example



```
public class Counter {
  private int count;                                        // a simple integer instance variable
  public Counter( ) { }                                     // default constructor (count is 0)
  public Counter(int initial) { count = initial; }                  // an alternate constructor
  public int getCount( ) { return count; }                            // an accessor method
  public void increment( ) { count++; }                                // an update method
  public void increment(int delta) { count += delta; }                 // an update method
  public void reset( ) { count = 0; }                                  // an update method
}
```

- instance variable
- methods
  - constructor
  - accessor
  - update

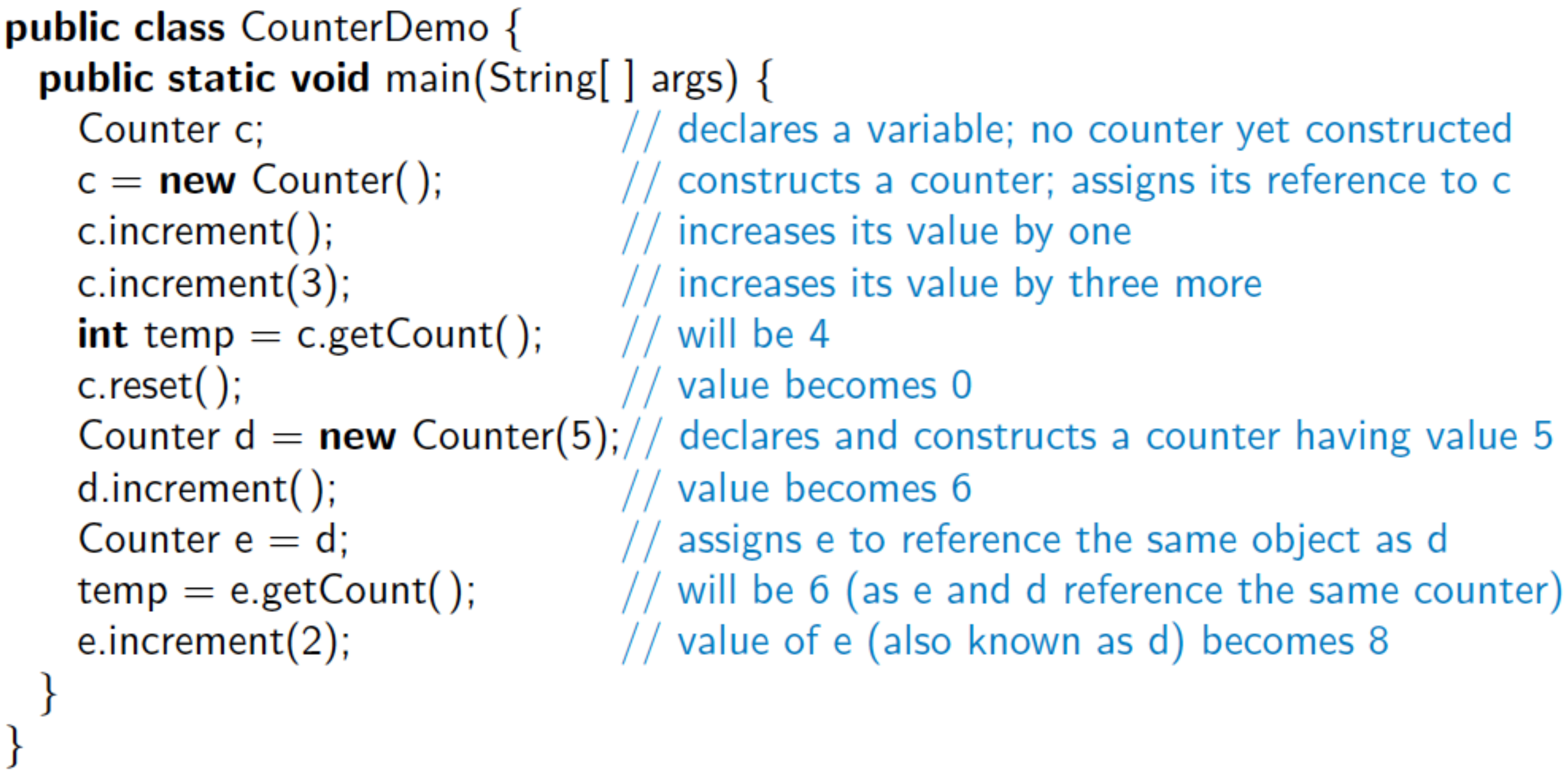
Creating and Using Objects

- Create an object by using the `new` operator followed by a call to a *constructor* for the desired class.

- Constructor:
  - a method that always shares the same name as its class
  - returns a reference to the newly created instance.

- Multiple constructors:
  - Empty constructor
  - Value constructors

Continued Example



```
public class CounterDemo {
  public static void main(String[ ] args) {
    Counter c;                    // declares a variable; no counter yet constructed
    c = new Counter( );           // constructs a counter; assigns its reference to c
    c.increment( );               // increases its value by one
    c.increment(3);               // increases its value by three more
    int temp = c.getCount( );     // will be 4
    c.reset( );                   // value becomes 0
    Counter d = new Counter(5);// declares and constructs a counter having value 5
    d.increment( );               // value becomes 6
    Counter e = d;                // assigns e to reference the same object as d
    temp = e.getCount( );         // will be 6 (as e and d reference the same counter)
    e.increment(2);               // value of e (also known as d) becomes 8
  }
}
```

Access Control Modifiers

- `public:`
  - designates that all classes may access
- `private`:
  - designates that access is granted only to code within that class.
- `static`
  - associates a variable/method with the class as a whole, rather than with each individual instance of that class

javadoc comments

- Comments
  - `/*   */`
  - `//`
- A style/format of commenting for auto-generation of documentation in html

```
/**
  */
```

- used for method headers and classes

Example

```
/**
 * returns the sum of two integers
 * @param x The first integer
 * @param y The second integer
 * @return int The sum of x+y
 */
int sum(int x, int y)
```

Casting – convert the type

- Assignment REQUIRES equal type

- Cast to change type

```
int x = 5;
double y = 1.2;
//y = x;
//x = y;
y = (double) x;
x = (int) y;
y = (double) x;
```

equals

- Compare primitives using Boolean operators:
  - <, >, <=, >=, ==
- Comparing objects:
  - Don’t use Boolean operators!
    - They check if two objects are the same
  - Use `.equals`
- Strings are objects in Java

```
String str1 = new String("one");
String str2 = new String("one");
str1 == str2; str1.equals(str2));
```

Strings Review

- Strings - `"a", "abc"`
- Characters – `'a'`
- Declaring String objects

```
String name;
String name = new String();
```

- Declaring String objects with initialization

```
String name = "Fred";
String name = new String("Fred");
```

`String` class methods

- `charAt(int index)`
  - Returns the character at the specified index
- `equals(String anotherString)`
  - Compares a string to a specified object
- `indexOf(char c)`
  - Returns the index value of the first occurrence of a character within the input string
- `indexOf(String str)`
  - Returns the index value of the first occurrence of a substring within the input string
- `length()`
  - Returns the number of characters in the input string
- `substring(int startIndex, int endIndex)`
  - Returns a new string that is part of the input string
- `toLowerCase()`
  - Converts all the characters to lower case
- `toUpperCase()`
  - Converts all the characters to upper case
- `String concat(String anotherString)`
  - Concatenates with anotherString and returns it

Parsing a line

- split a string into an array of Strings based on matching delimiter `delim`
- `String[] String.split(String delim)`

```
String s = "12,days,of,Christmas";
String[] tokens = s.split(",");
for (int i=0; i<tokens.length;i++) {
  System.out.println(tokens[i]);
}
```

Casting – String to primitives

```
String x = “5”;
String y = “1.2”;

double yDouble = Double.parseDouble(y)
int xInt = Integer.parseInt(x)
```

`Integer` and `Double` are examples of wrapper types

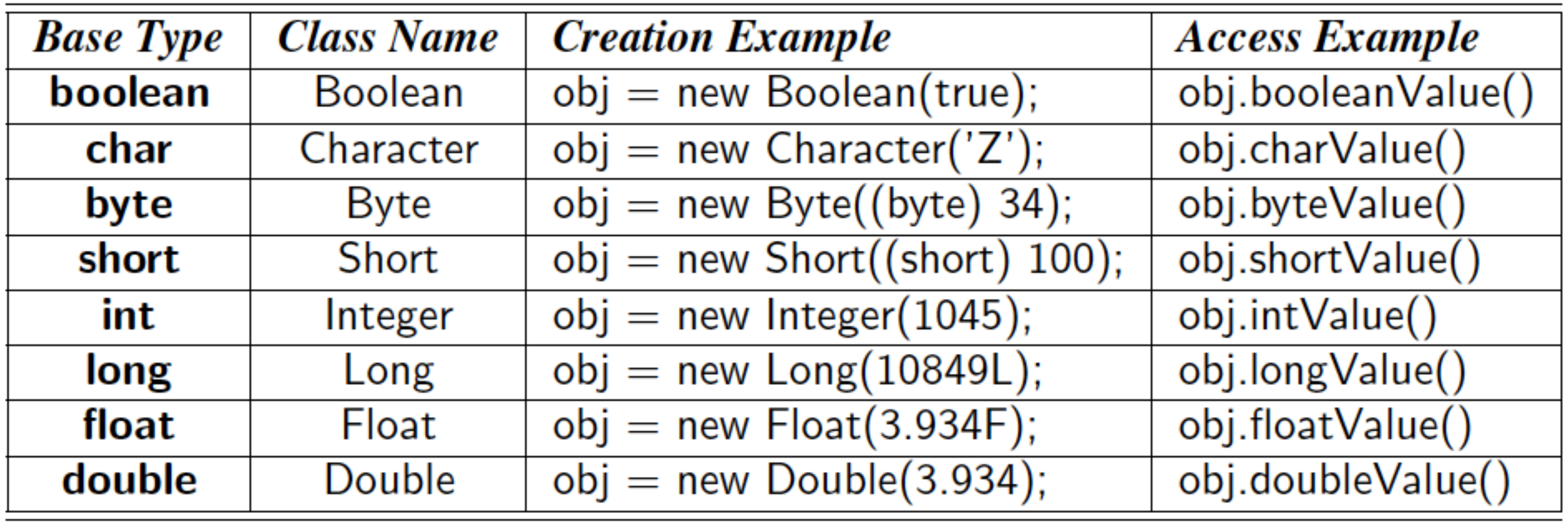
Wrapper Types

- Many data structures and algorithms in Java’s libraries only work with object types (not primitives)

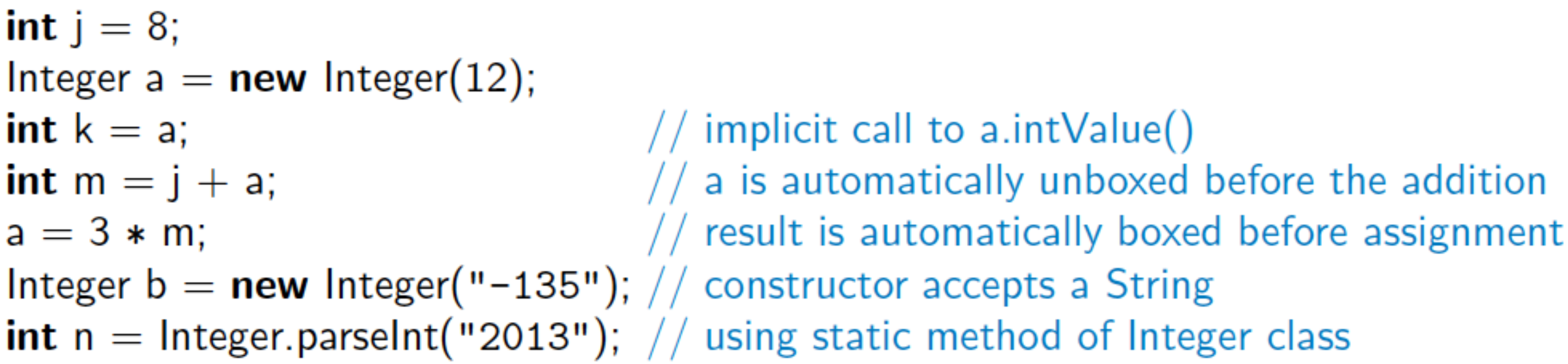
- To get around this obstacle, Java defines a *wrapper* class for each base type

- Implicitly converting between base types and their wrapper types is known as automatic *boxing* and *unboxing*.

Example



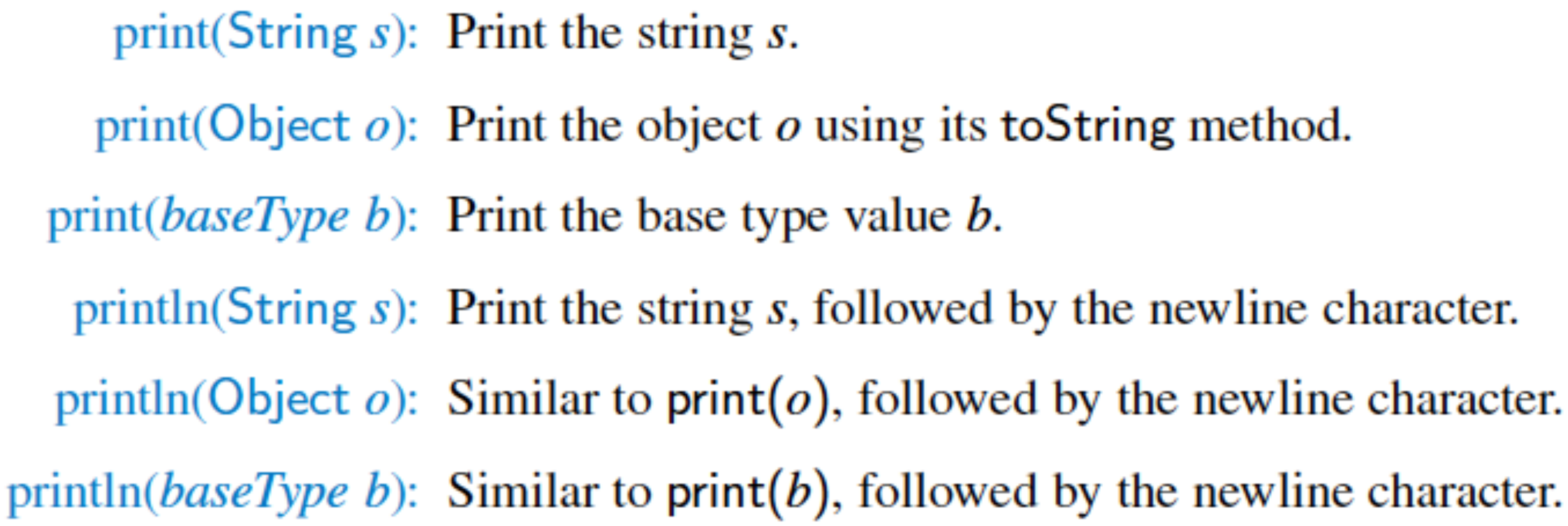
Base Type	*Class Name*	*Creation Example*	*Access Example*
boolean	Boolean	obj = new Boolean(true);	obj.booleanValue()
char	Character	obj = new Character('Z');	obj.charValue()
byte	Byte	obj = new Byte((byte) 34);	obj.byteValue()
short	Short	obj = new Short((short) 100);	obj.shortValue()
int	Integer	obj = new Integer(1045);	obj.intValue()
long	Long	obj = new Long(10849L);	obj.longValue()
float	Float	obj = new Float(3.934F);	obj.floatValue()
double	Double	obj = new Double(3.934);	obj.doubleValue()



```
int j = 8;
Integer a = new Integer(12);
int k = a;                          // implicit call to a.intValue()
int m = j + a;                      // a is automatically unboxed before the addition
a = 3 * m;                          // result is automatically boxed before assignment
Integer b = new Integer("-135");    // constructor accepts a String
int n = Integer.parseInt("2013");   // using static method of Integer class
```

Simple Output

- `System:`
  - Class in java for input/output
- `System.out:`
  - Variable in `System` class. What type of variable?
    - `static`



print(String *s*): Print the string *s*.

print(Object *o*): Print the object *o* using its toString method.

print(*baseType b*): Print the base type value *b*.

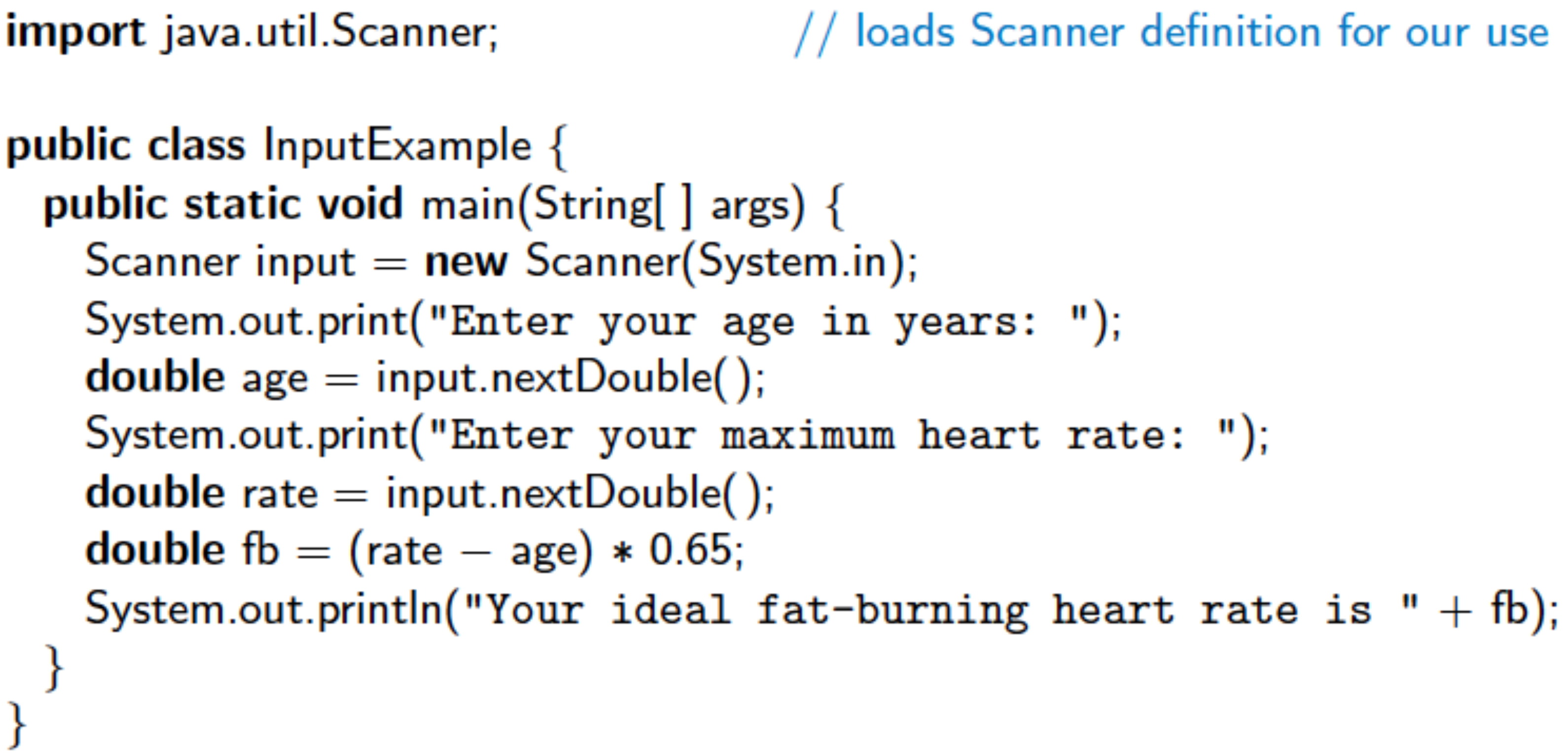
println(String *s*): Print the string *s*, followed by the newline character.

println(Object *o*): Similar to print(*o*), followed by the newline character.

println(*baseType b*): Similar to print(*b*), followed by the newline character.

Simple Input

- `System.in`
- `Scanner` object

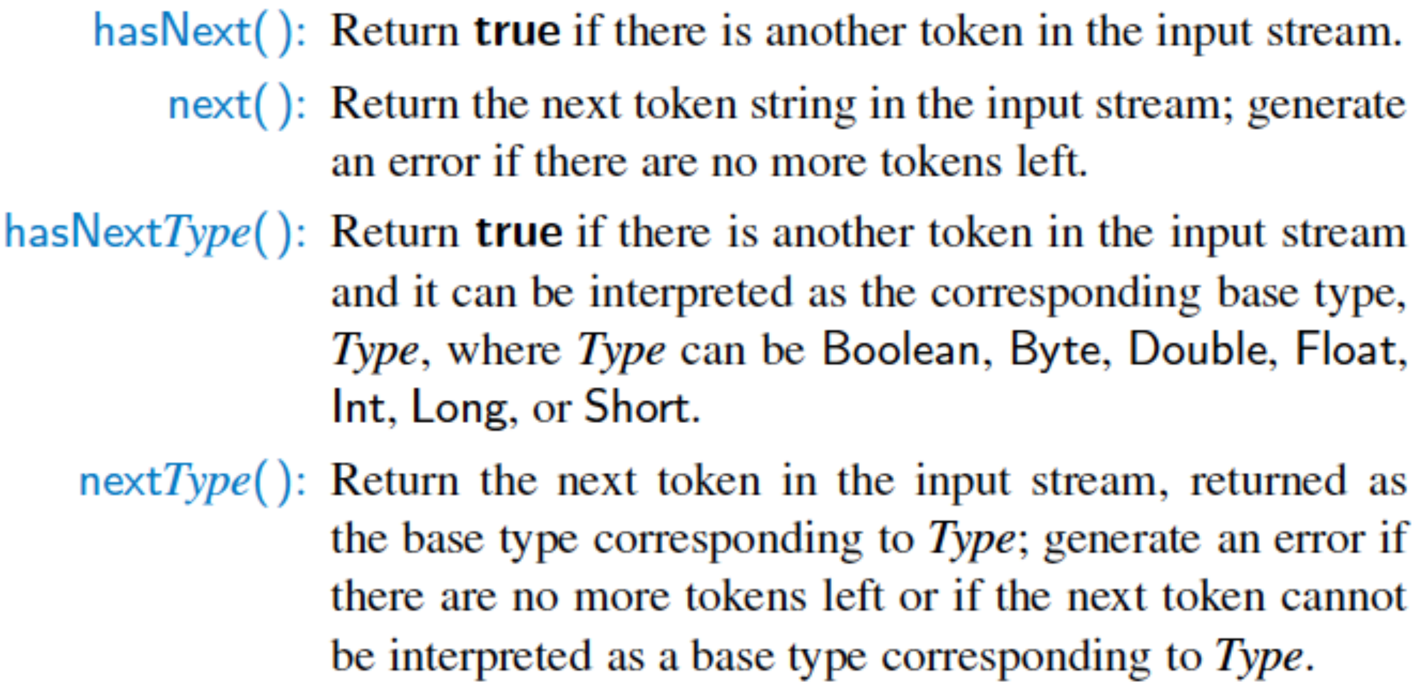


```
import java.util.Scanner;                    // loads Scanner definition for our use

public class InputExample {
  public static void main(String[ ] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter your age in years: ");
    double age = input.nextDouble( );
    System.out.print("Enter your maximum heart rate: ");
    double rate = input.nextDouble( );
    double fb = (rate − age) ∗ 0.65;
    System.out.println("Your ideal fat-burning heart rate is " + fb);
  }
}
```

`java.util.Scanner` Methods

- reads the input and divides it into tokens
- Tokens: strings separated by delimiters



hasNext(): Return **true** if there is another token in the input stream.

next(): Return the next token string in the input stream; generate an error if there are no more tokens left.

hasNext*Type*(): Return **true** if there is another token in the input stream and it can be interpreted as the corresponding base type, *Type*, where *Type* can be Boolean, Byte, Double, Float, Int, Long, or Short.

next*Type*(): Return the next token in the input stream, returned as the base type corresponding to *Type*; generate an error if there are no more tokens left or if the next token cannot be interpreted as a base type corresponding to *Type*.

Software Design Goals

- Robustness
  - software capable of error handling and recovery

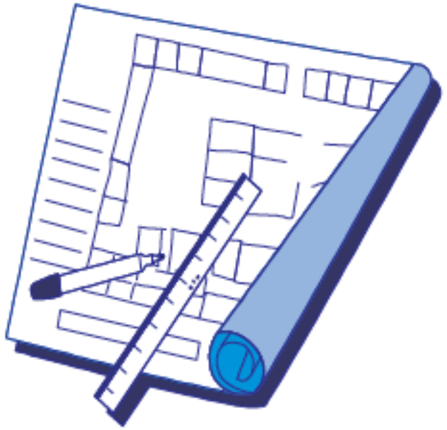
- Adaptability
  - software able to evolve over time and changing conditions (without huge rewrites)

- Reusability
  - same code is usable as component of different systems in various applications

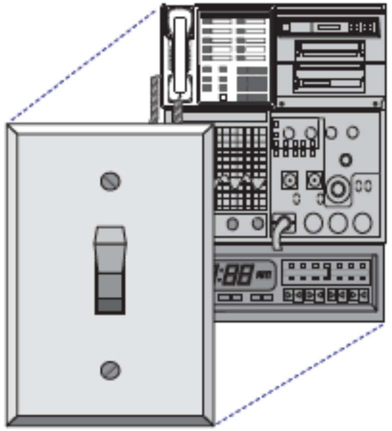
Object Oriented Programming Principles

- Modularity
- Abstraction
- Encapsulation

Modularity



Abstraction



Encapsulation

OOP Design

- Responsibilities/Independence: divide the work into different classes, each with a different responsibility and are as independent as possible
- Behaviors: define the behaviors for each class carefully and precisely, so that the consequences of each action performed by a class will be well understood by other classes that interact with it.

Class Definition

- Primary means for abstraction in OOP

- Class determines
  - the way state information is stored – via instance variables
  - a set of behaviors – via methods
- Class encapsulates
  - `private` instance variables
  - `public` accessor methods (getters)

Example

```
class Student {
  private String name;
  private int id;

  public Student(String name, int id) {
    this.name = name;
    this.id = id;
  }

  public String getName() {return name;}
  public int getId() {return id;}
}
```

Representing Objects

- What happens if we `System.out.println(obj)`?

```
Student s = new Student(”Ada Lee”, 1234);
System.out.println(s); //??
```

  - Prints location of the object in memory

- `toString()`
  - Special method in a class that provides a way to customize printing objects
  - returns a `String` representation of the instance object that is used by `System.out.println`
  - `public String toString()`

Student

```
class Student {
  private String name;
  private int id;
  // constructor and getters not shown

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

Inheritance

- Allow a new class to be defined based on an existing class
  - Existing: base, super or parent class
  - New: subclass or child class

- Keyword `extends`

```
    class CSStudent extends Student{ … }
```

- `CSStudent` **inherits all** `public` **and** `protected` **instance variables and methods of** `student`

Constructors

- Constructors are never inherited

- A subclass may invoke the superclass constructor via a call to `super` with the appropriate parameters

- If calling `super`, it must be in the first line of the subclass' constructor

- If no explicit call to `super`, then an implicit call to the zero-parameter `super()` will be made

CSStudent

```
class CSStudent extends Student{
  private boolean isMajor;
  public CSStudent(String name, int id, boolean isMajor){
    super(name, id);
    this.isMajor = isMajor;
  }
  public boolean getIsMajor() {return isMajor;}
}
CSStudent s1 = new CSStudent(”Adam Po", 1111, true);
CSStudent s2 = new CSStudent("Di Xu", 2222, false);
System.out.println(s1);
System.out.println(s2);
```

Output

```
Adam Po 1111
Di Xu 2222
```

`public` versus `default`

- What access modifier is used when you don't put any?
    - `class Student`
    - `Student(String name, int id)`
- default = package
    - visible within same package (directory)
    - not the same as `public`
- Constructor and class modifiers match

Source Code Organization

- Each project under its own subdirectory
  - directory name = project name
  - A1, A2, …
- One class per file - `public`
- name of the file matches class name
- `Driver.java`
- compiling just `Driver.java` usually compiles all

File I/O

1. import packages

```
import java.io.*
import java.util.*
```

2. Create a new `Scanner` object linked to the file we want to read

```
Scanner input = new Scanner(new File(<filename>));
```

3. Use `hasNextLine()` and `nextLine()` methods to read line by line until done

```
while(input.hasNextLine()) {
  String line = input.nextLine();
  ...
}
```

4. Close

```
input.close;
```

Exceptions – way to deal with unexpected events during execution

- Unexpected events:
  - unavailable resource
  - unexpected input
  - logical error

- Exceptions are objects that can be *thrown* by code expecting to encounter it

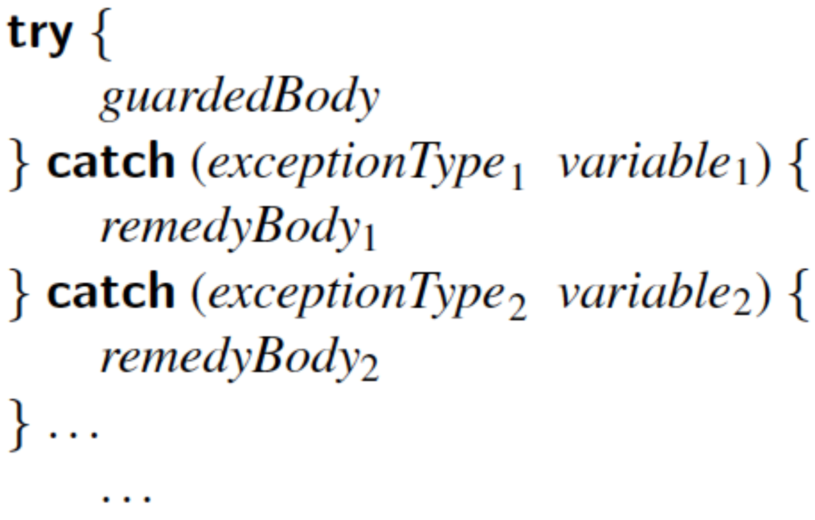
- An exception may also be *caught* by code that will handle the problem

Catching Exceptions

- Exception handling

  `try-catch`

- An exception is caught by having control transfer to the matching `catch` block



```
try {
    guardedBody
} catch (exceptionType1  variable1) {
    remedyBody1
} catch (exceptionType2  variable2) {
    remedyBody2
} ...
    ...
```

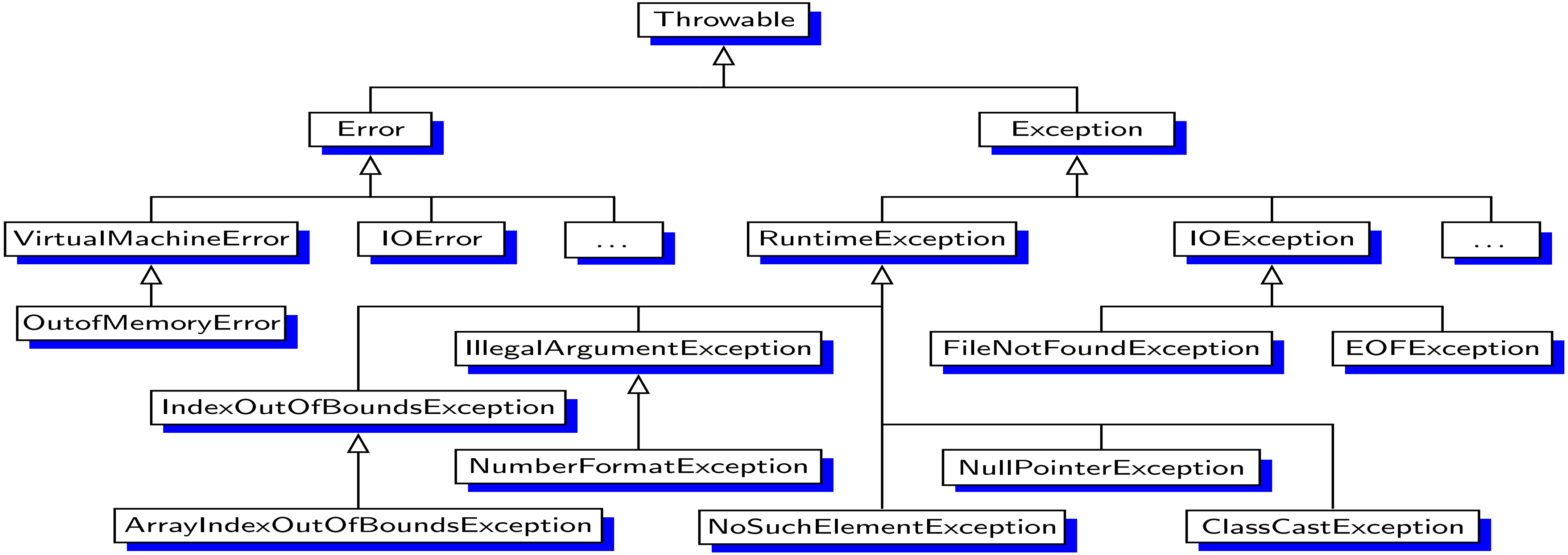
- If no exception occurs, all `catch` blocks are ignored

Throwing Exceptions

- An exception is thrown
  - implicitly by the JVM because of errors
  - explicitly thrown by code
- Exceptions are objects
  - throw an existing/predefined one
  - make a new one
- Method signature – `throws`

```
public static int parseInt(String s) throws
NumberFormatException
```

Java’s Exception Hierachy



What you should know/review

- variables
- expressions
- operators
- methods
  - parameters
  - return value
- conditionals
- `for`/`while` loops
- class design and object construction
  - instance variables
  - constructor
  - getters/setters
  - class methods
  - `new`
- arrays
- arrays of objects
- `String`

What you don’t know

- Read the manuals/references
    - Unix commands (flags, usage, examples)
    - Java methods (parameters/overloading)
- Google – but with judgement
- Trial-and-Error is a fundamental method of problem-solving
- The ability to tinker is a fundamental engineering/CS skill