CS 5004

LECTURE 2

METHODS, ENUMS, EXCEPTIONS

KEITH BAGLEY

SPRING 2022

Northeastern University
Khoury College of Computer Sciences

440 Huntington Avenue ▪ 202 West Village H ▪ Boston, MA 02115 ▪ T 617.373.2462 ▪ khoury.northeastern.edu

AGENDA

- Housekeeping
- Methods
- Enumerations
- Exception Handling
- Packages for Organization

HOUSEKEEPING

- Individual work MUST be individual work
  - This includes quizzes, homework, exams!
  - Labs done during recitation may be partially “group think” as your TA facilitates an active working session
  - I will be using Moss for code similarity analysis, other forensic analysis, and other means to provide an equitable environment
  - If you are caught cheating, you will fail the assignment and then be awarded with an “F” for the course

ACADEMIC INTEGRITY

Collaboration and academic integrity

You may not collaborate with anyone on any of the exams or quizzes. You may not use any electronic tools, including phones, tablets, netbooks, laptops, desktop computers, etc. If in doubt, ask a member of the course staff.

Some but not all homework assignments will be completed with an assigned partner, and some may involve a larger team (TBD). You must collaborate with your assigned partner or team, as specified, on homework assignments. You may request help from any staff member on homework. (When you are working with a partner, we strongly recommend that you request help with your partner.) You may use the Piazza bulletin board to ask questions regarding assignments, so long as your questions (and answers) do not reveal information regarding solutions. **You may not get any help from anyone else on a homework assignment; all material submitted must be your own.** If in doubt, ask a member of the course staff.

Providing illicit help to another student is also cheating, and will be punished the same as receiving illicit help. It is your responsibility to safeguard your own work.

Students who cheat will be reported to the university's office on academic integrity and penalized by the course staff, at our discretion, up to and including failing the course. You will also be reported to the Khoury academic integrity committee, **which imposes harsher penalties than the University does (including the possible loss of co-op experiences)**.

If you are unclear on any of these policies, please ask a member of the course staff.

- I've highlighted this section of our syllabus in Canvas in case you missed it.

REVIEW: CLASSES & OBJECTS

- Classes are the “blueprint” or the “cookie cutter”
  - They describe the recipe for how to make objects
- Objects are the “thing” made (instances). They are the cookies.
- Objects have
  - Attributes (value/data)
  - Behavior (operations/methods)
  - State (a condition it’s in)
  - Unique identity (self existence)

PACKAGES

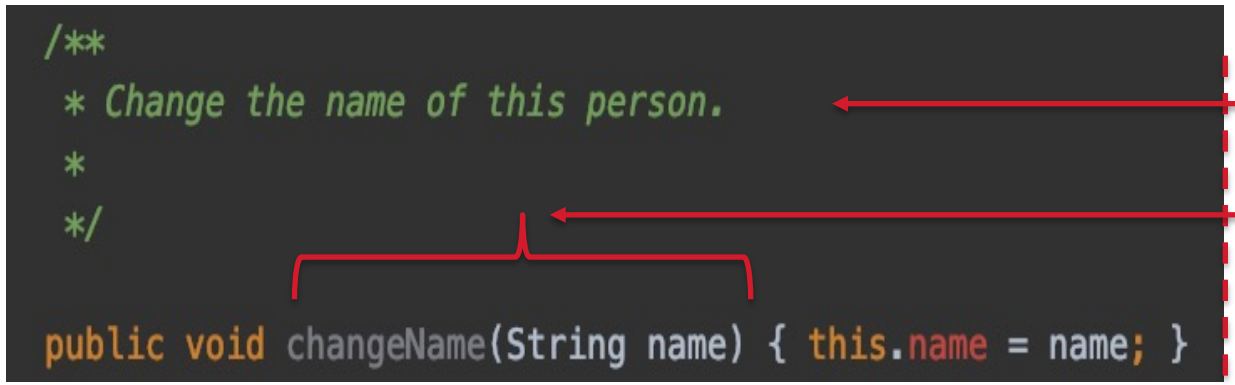
- Packages are used in organizing our code
  - They are used to group related classes
  - Introduce a new scope to avoid name conflicts
  - Should be written in lowercase to avoid name conflicts with classes
  - Use the file system to store in directories
- We've already imported existing packages from the Java library (e.g. org.junit)
- We'll be using user-defined packages today for our code
- Can create our own packages via the package keyword

```
package mypack;
class MyPackageClass {
```

WHAT ARE METHODS?

- We covered encapsulation last week: the grouping of data & functions together into one coherent unit
  - The *functions* are called *methods* (or member functions in some languages)
  - The *data* is called *instance variables**
- Methods are part of a coherent whole
  - That “whole” is the class the method belongs to
- Each method has a (single) purpose
- Methods have a signature that tells users the name, parameters and return type
  - NB: A return type of “void” means the method returns nothing
- Methods have a method body which defines the code that is executed when invoked

METHODS: SIGNATURE



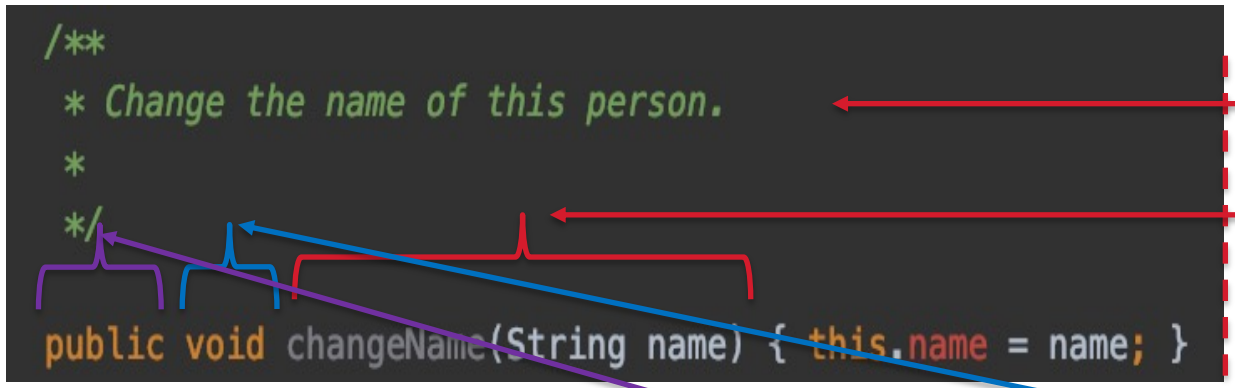
Javadoc comments w/ purpose, return values, parameters

Signature

- `changeName(String)` is the signature in this example

Strictly speaking, the **signature** of a Java method is name of the **method** and its list of **parameter types**

METHODS: SIGNATURE



Javadoc comments w/ purpose, return values, parameters

Signature
- `changeName(String)` is the signature in this example

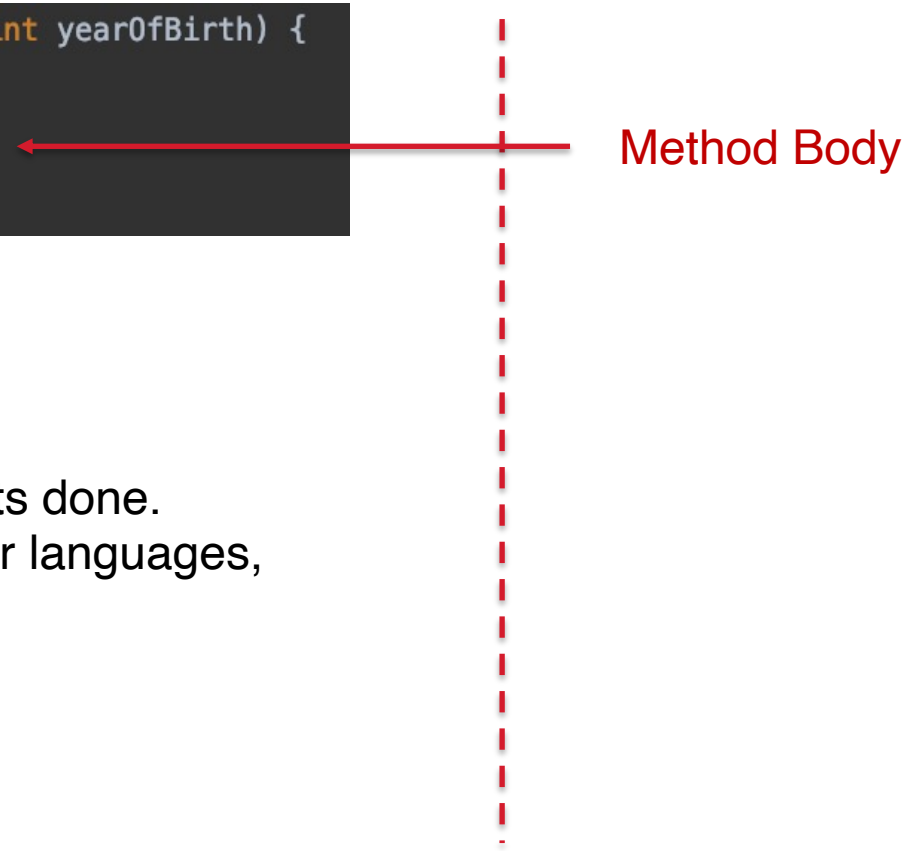
Return type

Access Specifier

Strictly speaking, the **signature** of a Java method is name of the **method** and its list of **parameter types**
- You may hear some people including the return type and access specification in their description of the signature. To be precise, those extra elements are **not** part of the signature

METHODS: BODY

```
public Person(String firstName, String lastName, int yearOfBirth) {
  this.firstName = firstName;
  this.lastName = lastName;
  this.yearOfBirth = yearOfBirth;
}
```



The method body is where the work gets done. Methods are similar to functions in other languages, in that they have their own scope

METHODS INTRODUCE A NEW SCOPE

- Variables introduced at method scope take “precedence” over enclosing scopes if the names are the same
  - Remember our instance variables are at “class scope”. Use the `this` keyword to disambiguate if needed

"STATIC" METHODS ARE CLASS-LEVEL

- The keyword static is used to create a "class method" that does not require an instance to call it. You may have noticed in some of the readings and our examples:
  - `public static void main(String [] args)`
  - `Math.pow(radius, 2);`
- Typically, we need an object to call a method. Static methods can be called with only the class.
- Static can be applied to data as well. Static data is essentially "class variables" (rather than instance variables).
  - Only ONE copy of the data is available for the entire class. Each instance shares that single copy.
  - Useful for class-specific constants

WRITING METHODS: RECIPE

- What is the method for? Document this in high-level terms
- Covert the description into a method signature
- Create your tests
- Implement the method functionality

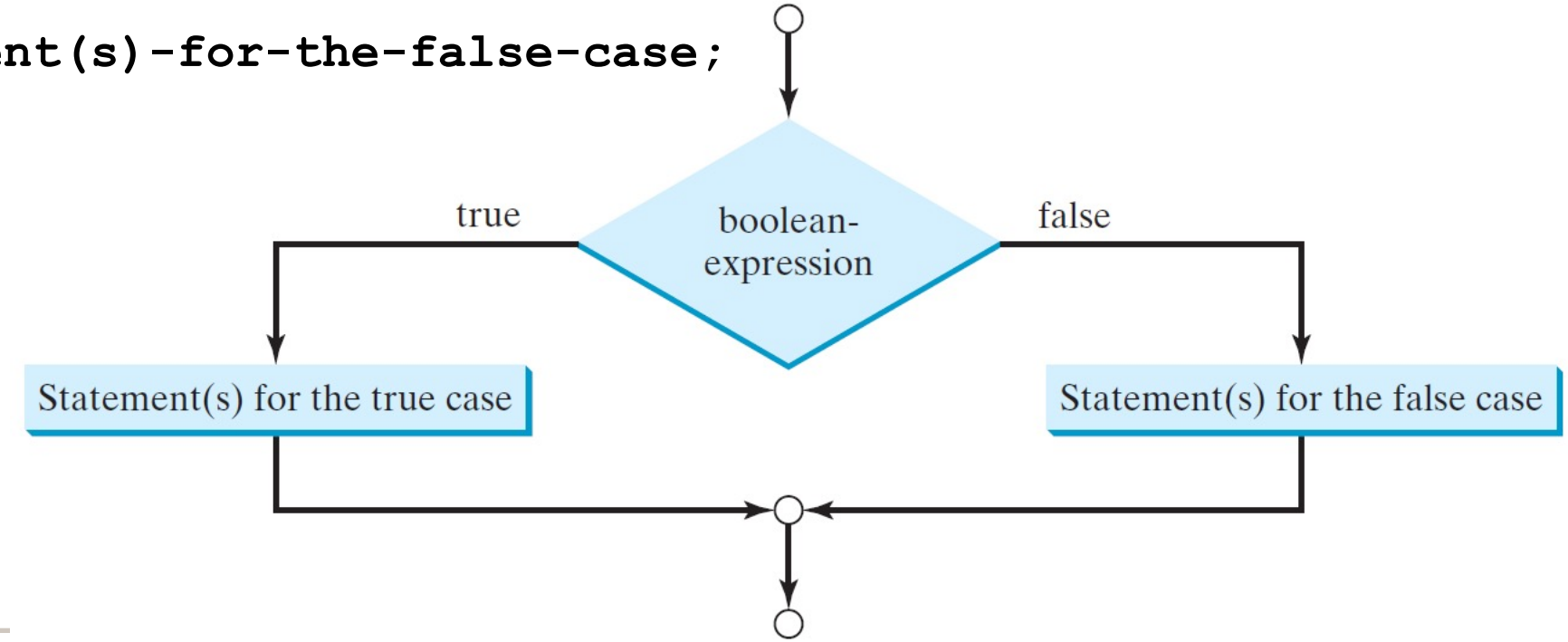
- Ask:
  - What does the object know already?
  - What does the object do?
  - What does the method need to do the work?

SIDEBAR: DECISION MAKING

- Of course, methods may entail conditional statements for decision making
- In Java, can use:
  - If
  - If-else
  - Switch statement

SIDEBAR: IF STATEMENT

```
if (boolean-expression) {
  statement(s)-for-the-true-case;
}
else {
  statement(s)-for-the-false-case;
}
```



SIDEBAR: SWITCH STATEMENT

The switch-expression must yield a value of char, byte, short, or int type and must always be enclosed in parentheses.

The value1, ..., and valueN must have the same data type as the value of the switch-expression. The resulting statements in the case statement are executed when the value in the case statement matches the value of the switch-expression. Note that value1, ..., and valueN are constant expressions, meaning that they cannot contain variables in the expression, such as 1 + x.

```
switch (switch-expression) {
  case value1:  statement(s)1;
          break;
  case value2: statement(s)2;
          break;
  …
  case valueN: statement(s)N;
          break;
  default: statement(s)-for-default;
}
```

SIDEBAR: SWITCH STATEMENT

```python
def calc_area(choice):
    ''' function do_area
        Input: choice - the type of shape selected by the user
        Returns: area of a shape, depending on the type chosen
        Does: Asks the user for shape's dimensions to cacluate area
    '''
    if choice == 'S':
        length = float(input('Enter the square length: '))
        return round((length * length),2)
    elif choice == 'C':
        radius = float(input('Enter the circle radius: '))
        return round((PI * radius**2),2)
    elif choice == 'T':
        base = float(input('Enter the triangle base: '))
        height = float(input('Enter the triangle height: '))
        return round((0.5 * base * height),2)
    else:
        print('I do not know that shape!')
    return 0
```

```java
public double calcArea(char choice) {
  switch(choice) {
    case 'S':
      // calc and return the area of Square
      break; // talk about this if return statement is used - not needed
    case 'C':
      // calculate and return area of circle
      break;
      // etc.
    default:
      // do something useful here
  }
  return 0;
}
```

METHODS: OVERLOADING

- A class may have two or more methods with the same name BUT DIFFERENT SIGNATURES.
- This is useful when:
  - there is a slight variation in the incoming data for a method or,
  - a slight variation in behavior due to the incoming data

METHODS: OVERLOADING

```java
/**
 * Car constructor
 * Create a new instance of a Car.
 */

public Car(String make, String model) {
  this.make = make;
  this.model = model;
  year = 1986; // did not need to fully qualify. Implicit "this" here
  price = 6000.0;
}

/**
 * Car constructor overload
 * Create a new instance of a Car.
 */

public Car(String make, String model, int year, double price) {
  this.make = make;
  this.model = model;
  this.year = year; // this time, we DO need to disambiguate with "this"
  this.price = price; // same here
}
```

Overloaded Constructor for Car class

METHODS: OVERLOADING

```java
/**
 * Car constructor
 * Create a new instance of a Car.
 */

public Car(String make, String model) {
  this.make = make;
  this.model = model;
  year = 1986; // did not need to fully qualify. Implicit "this" here
  price = 6000.0;
}

/**
 * Car constructor overload
 * Create a new instance of a Car.
 */

public Car(String make, String model, int year, double price) {
  this.make = make;
  this.model = model;
  this.year = year; // this time, we DO need to disambiguate with "this"
  this.price = price; // same here
}
```

What's different?
1. The Signatures. That's mandatory
2. The Bodies. (makes sense)

What's the same?
1. Same name
2. Same scope

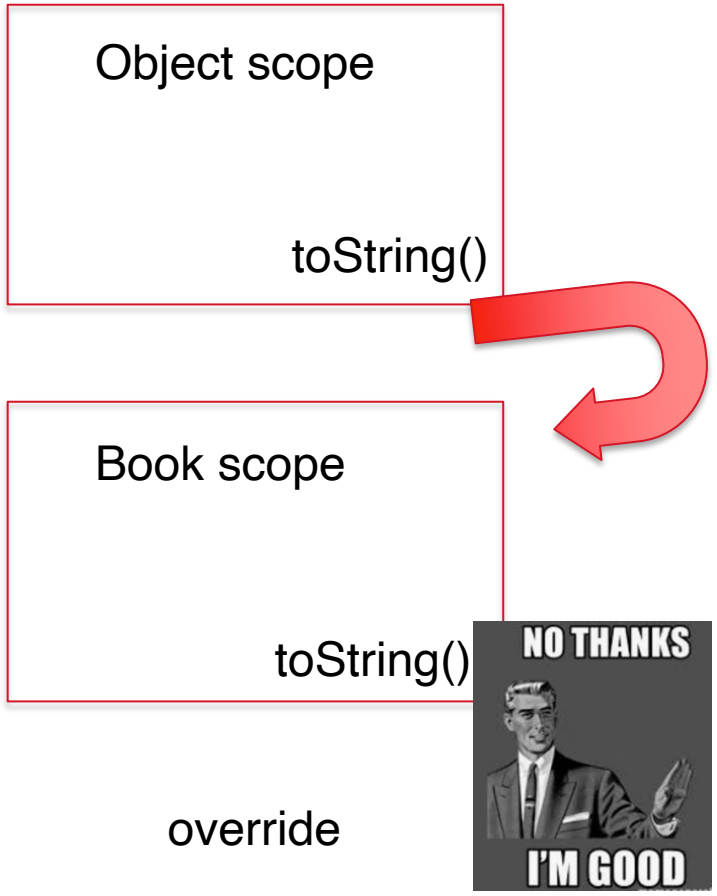
METHODS: OVERLOADING - USE

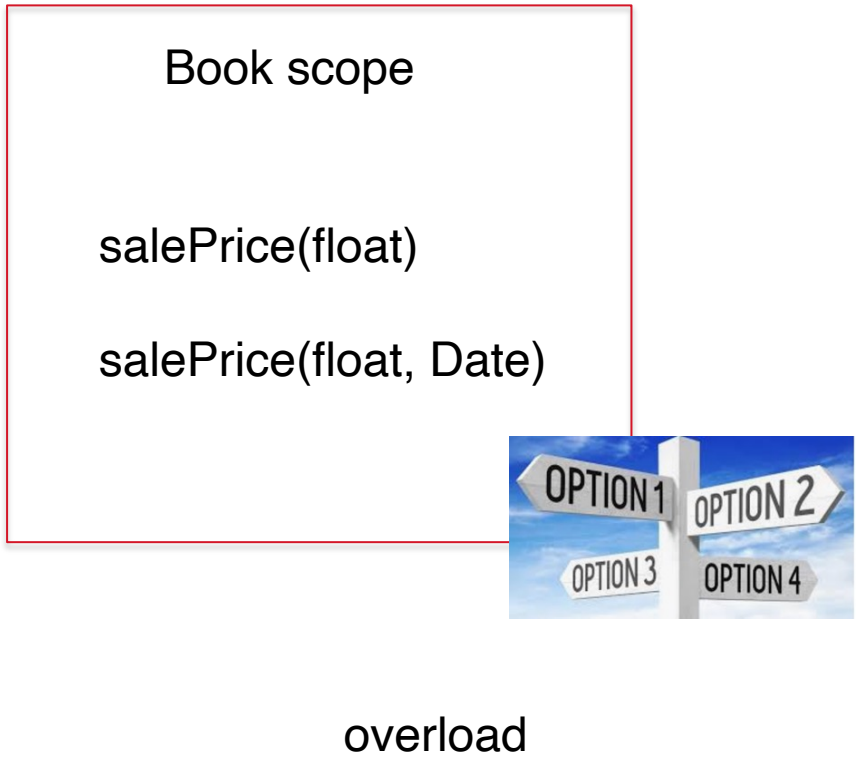
```
static public void main(String [] args ) {
    Car c = new Car( make: "Nissan",  model: "Sentra");
    Car c2 = new Car( make: "Ford",  model: "Escort",  year: 2010,  price: 15500.0);
    CarFeature item = new CarFeature( name: "heated seats",  price: 250.0);
    c.addFeature(item);
    System.out.println(c.getPrice());
    System.out.println(c2.getPrice());
  }
}
```

2-argument constructor

4-argument constructor

CLARIFY: OVERRIDE VS. OVERLOAD





CLARIFY: OVERRIDE VS. OVERLOAD

```java
/**
 * Return a formatted string that contains the information
 * of this object. The string should be in the following format:
 *
 * Title: [title of the book]
 * Author: [first-name last-name]
 * Price: [Price as a decimal number with two numbers after decimal]
 *
 * @return the formatted string as above
 */
public String toString() {
  String str;

  str = "Title: "+ this.title + "\n" +
        "Author: "+this.author + "\n";
  str = str + String.format("Price: %.2f",price);

  return str;
}
```

Override toString()

```java
public float salePrice(float discount) throws IllegalArgumentException {

  if (discount<0) {
    throw new IllegalArgumentException("Discount cannot be negative");
  }

  return this.price - (this.price * discount) / 100;
}

public float salePrice(float discount, Date expiration) throws IllegalArgumentException {

  if (discount<0) {
    throw new IllegalArgumentException("Discount cannot be negative");
  }
  Date today = new Date();
  if (today.before(expiration)) {
    return this.price - (this.price * discount) / 100;
  }
  else{
    return this.price;
  }
}
```

Overload salePrice()

LET'S PRACTICE SOME OF THIS WITH CIRCLE

Using our Circle class from last week, modified

ENUMERATIONS

An enumeration (enum) is a list of named constant values

WHO-WHAT: VARIATION IN DATA

- Books can be in different formats
  - For this example, assume catalog just needs to know if the book is hardcover, kindle, etc.
  - No variation in behavior. Each kind of Book behaves the same, other than perhaps how it represents itself
  - **enums** (or perhaps method overloading) is the best option here
    - Finite set of discrete values
    - Localized point of selection (1 or 2 methods) to avoid ripple effect
    - Compile-time enforcement of proper values

```
public enum TypeOfBook {HARDCOVER,PAPERBACK,KINDLE}
```

We can use `switch` in the above `toString` as follows:

```
public String toString() {
  String str;

  str = "Title: " + this.title + "\n" +
        "Author: " + this.author + "\n" +
        "Type: ";
  switch (bookType) {
    case PAPERBACK:
      str = str + "Paperback";
      break;
    case HARDCOVER:
      str = str + "Hard Cover";
      break;
    case KINDLE:
      str = str + "Kindle";
      break;
  }

  str = str + "\n";
  str = str + String.format("Price: %.2f", price);

  return str;
}
```

ENUMERATED TYPES

- An enumerated type (enum in Java) is a kind of “named constant” in which you provide a short list of possible values upon specification
  - Typically, we are not concerned about the *underlying value* how the enumeration is represented
  - We do care about the *conceptual* value
  - We may care about *sequencing* (predecessor/successor relationship)

ENUMS (ENUMERATED TYPES)

```java
/**
 * Enum example using calendar CalendarEvent class to track events and days they occur
 */

public class CalendarEvent {
  // create a set of values
  enum DAY { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };

  private DAY dayOfEvent;
  private String eventName;

  /**
   * Simple constructor to create an event on a given day
   * @param dayOfEvent
   * @param eventName
   */

  public CalendarEvent(DAY dayOfEvent, String eventName) {
    this.dayOfEvent = dayOfEvent;
    this.eventName = eventName;
  }

  /**
   * Method to answer when the event is being held
   * @return the day of the week the event is
   */

  public DAY whenIsEvent() {
    return this.dayOfEvent;
  }
```

enum
Set of related values

We don't really care about the "underlying value"

ENUMS (ENUMERATED TYPES)

```
 * Method to answer the name/description of the event
 * @return the event name. future versions may swap in a description
 */

public String whatIsEvent() {
  return this.eventName;
}

/**
 * Method answers true/false depending on if the event is scheduled for the weekend
 * @return boolean indicating if the event happens on a weekend day
 */

public boolean occursOnWeekend() {
  if(this.dayOfEvent == DAY.SUNDAY || this.dayOfEvent == DAY.SATURDAY) {
    return true;
  }
  return false;
}

/**
 * Method answers true/false depending on if the event is scheduled for the weekend,
 * using the compareTo() method that enums implement
 * @return boolean indicating if the event happens on a weekend day
 */

public boolean occursOnWeekendV2() {
  if(this.dayOfEvent.compareTo(DAY.SATURDAY) >= 0 ) { // 0 == equals, -1 == less than
    return true;                                      // 1 == greater than
  }
  return false;
}
```

Using enum

Most times we care about the conceptual value, not the value representation. Is Saturday 6? Who cares? We want to know about the weekend events

ENUMS & SWITCH STATEMENT

- When the list of choices in an enumeration is more than two or three, it is often cleaner to use a multiway branching statement (switch) rather than a set of multiple if-else pairs

SWITCH WITH ENUMS

```java
/**
 * @Override toString
 * @return a string representation of the CalendarEvent, with a snarky message
 */

@Override
public String toString() {
  String result;
  switch(this.dayOfEvent) {
    case MONDAY:
    case TUESDAY:
      result = "Lab and Lecture! Yay!";
      break;  // this is needed to short-circuit flow-down
    case FRIDAY:
      result = "Ba da ba ba ba - I'm lovin' it!";
      break;
    case SATURDAY:
    case SUNDAY:
      result = "Livin' for the weekend!";
      break;
    default:
      result = "Drake: I'm not Kiki & I don't love this day of the week";
  }
  return this.eventName + " on " + this.dayOfEvent + " -> " + result;
}
```

Switch example, using enumerations

SWITCH WITH ENUMS

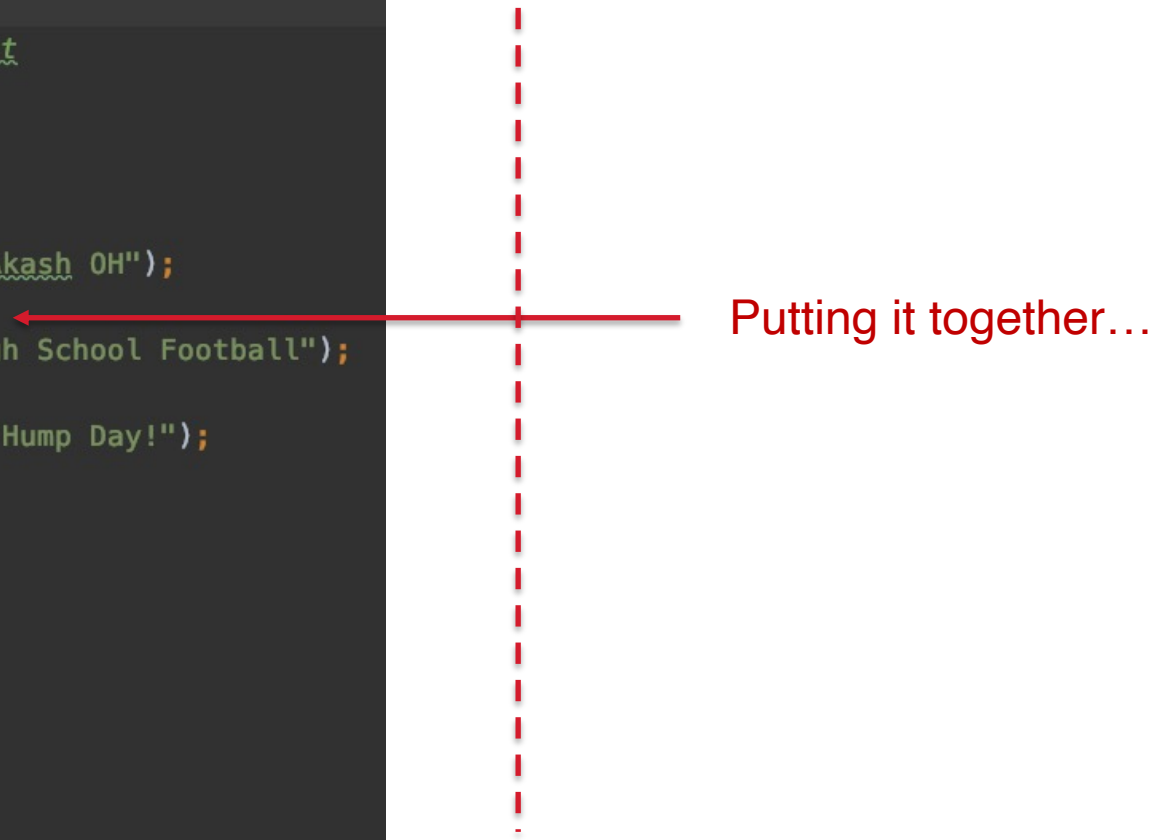
```
/**
 * Simple test driver here, rather than writing a separate Main
 * class for this example
 * @param args arguments for the static main() entrypoint
 */

static public void main(String [] args ) {
  CalendarEventV2 officeHours =
          new CalendarEventV2(DAY.SATURDAY,  eventName: "Akash OH");
  CalendarEventV2 footballGame =
          new CalendarEventV2(DAY.FRIDAY,  eventName: "High School Football");
  CalendarEventV2 humpDay =
          new CalendarEventV2(DAY.WEDNESDAY,  eventName: "Hump Day!");

  System.out.print("Friday Night Lights. Event is: ");
  System.out.println(footballGame);

  System.out.print("Akash's office hours: ");
  System.out.println(officeHours);

  System.out.print("Wednesday: ");
  System.out.println(humpDay);
}
```



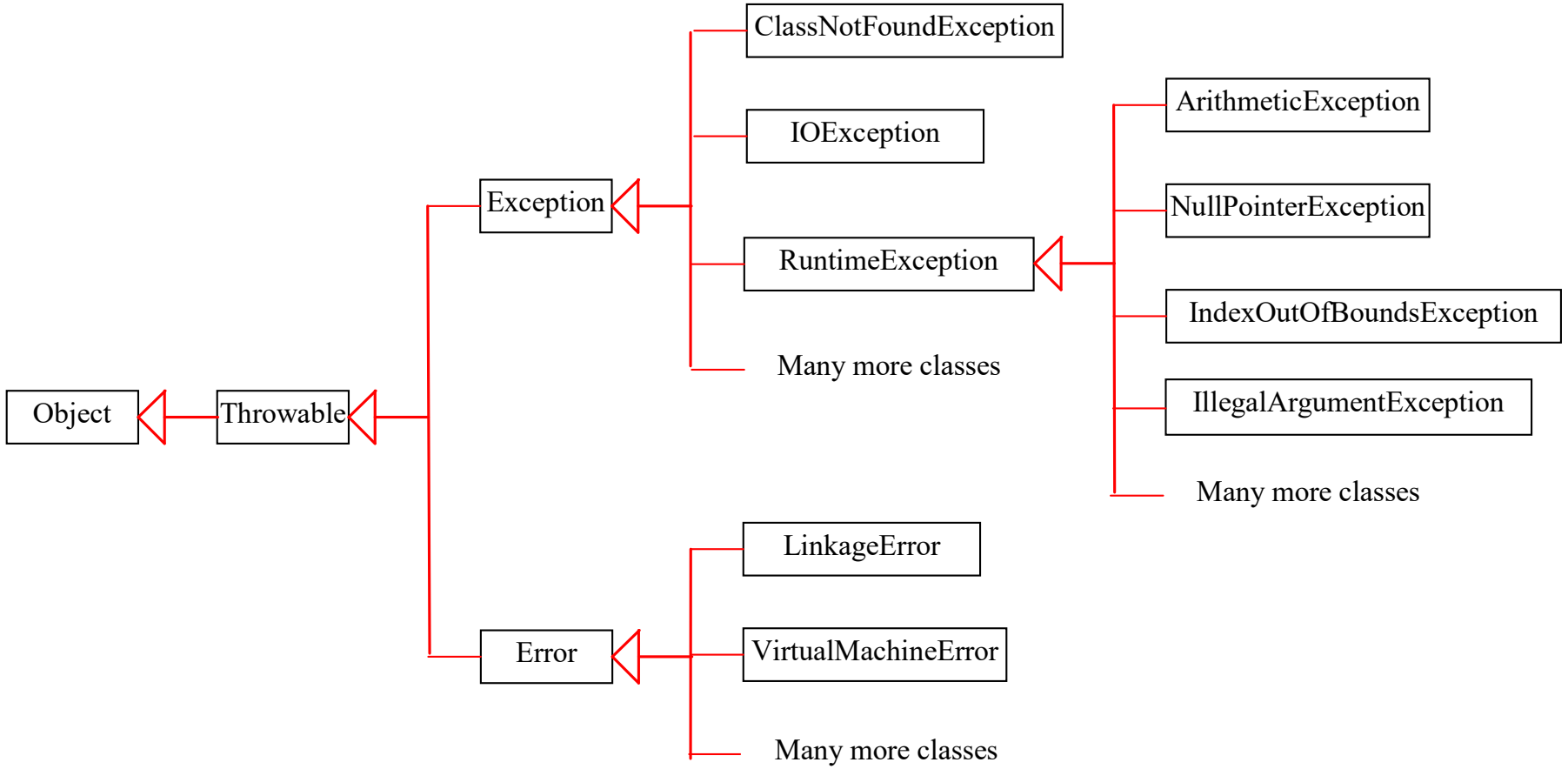
LET'S ADD AN ENUM FOR CIRCLE

Using our Circle class from last week, modified

EXCEPTIONS & EXCEPTION HANDLING

- In your rooms, consider the set of tests that would exercise your SimpleFractions
  - Boundary conditions
  - Fuzz tests?
  - Illegal arguments

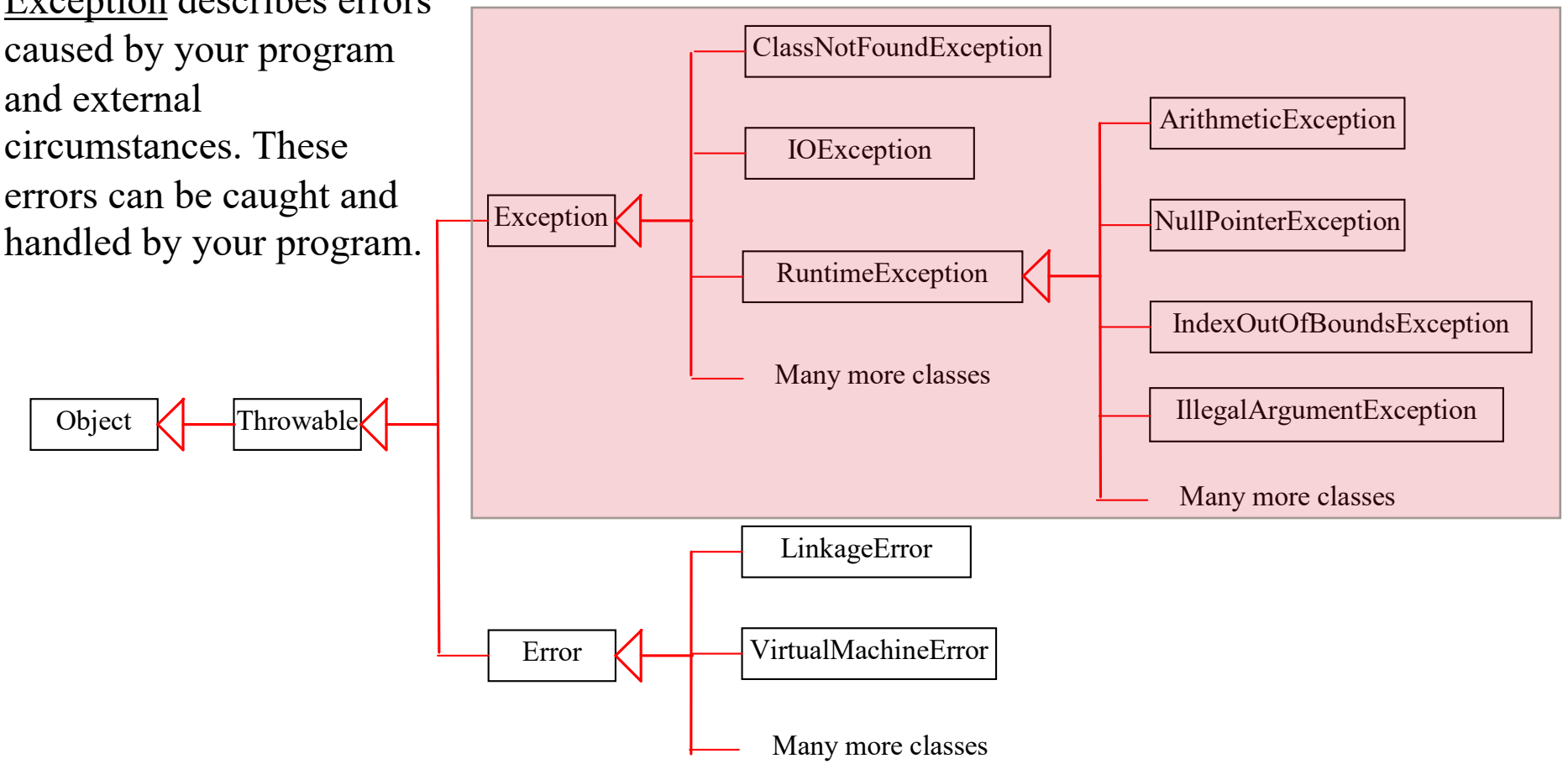
EXCEPTION TYPES



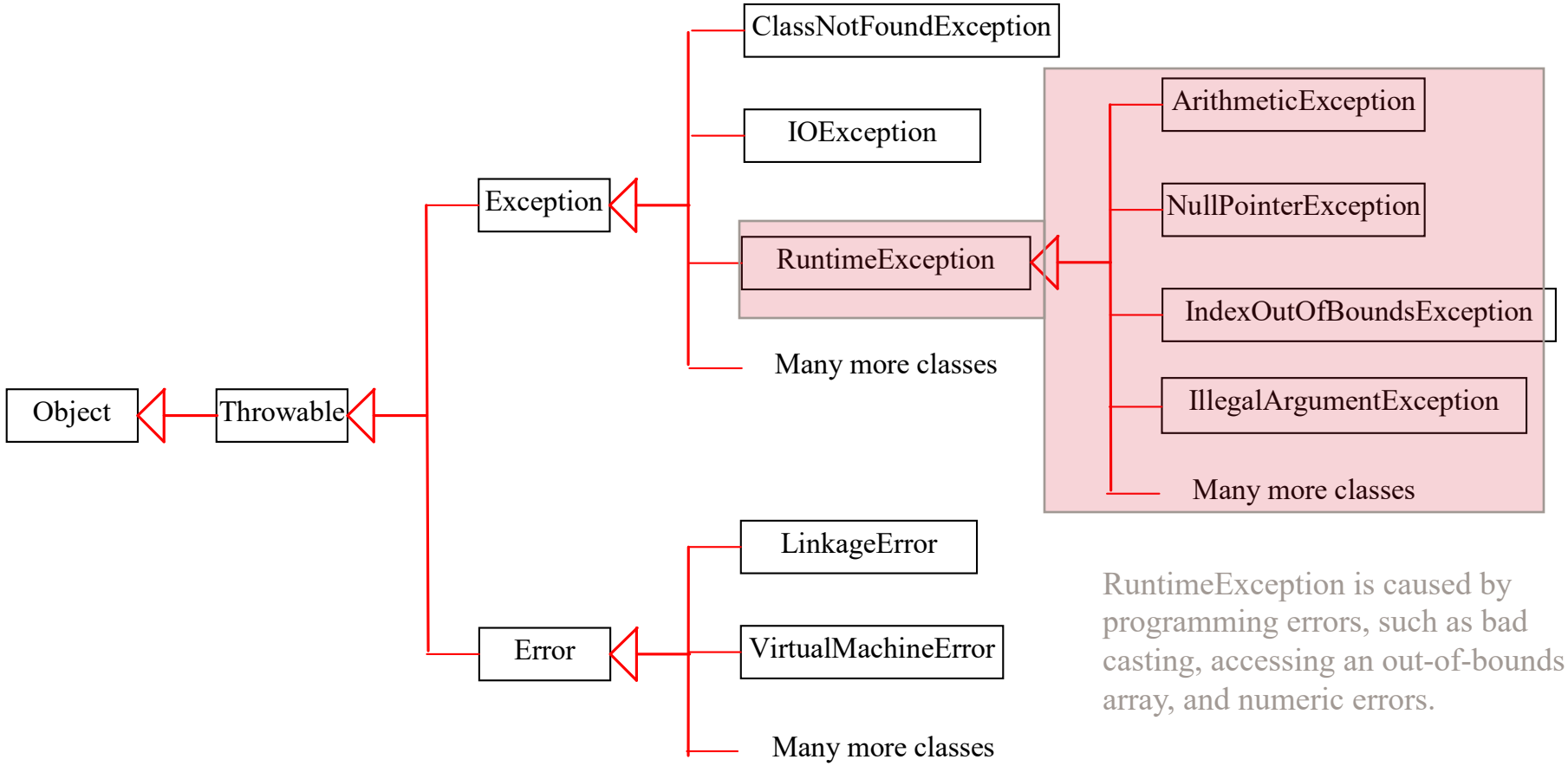
Content from Liang - 2019

EXCEPTIONS

Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.



RUNTIME EXCEPTIONS



RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

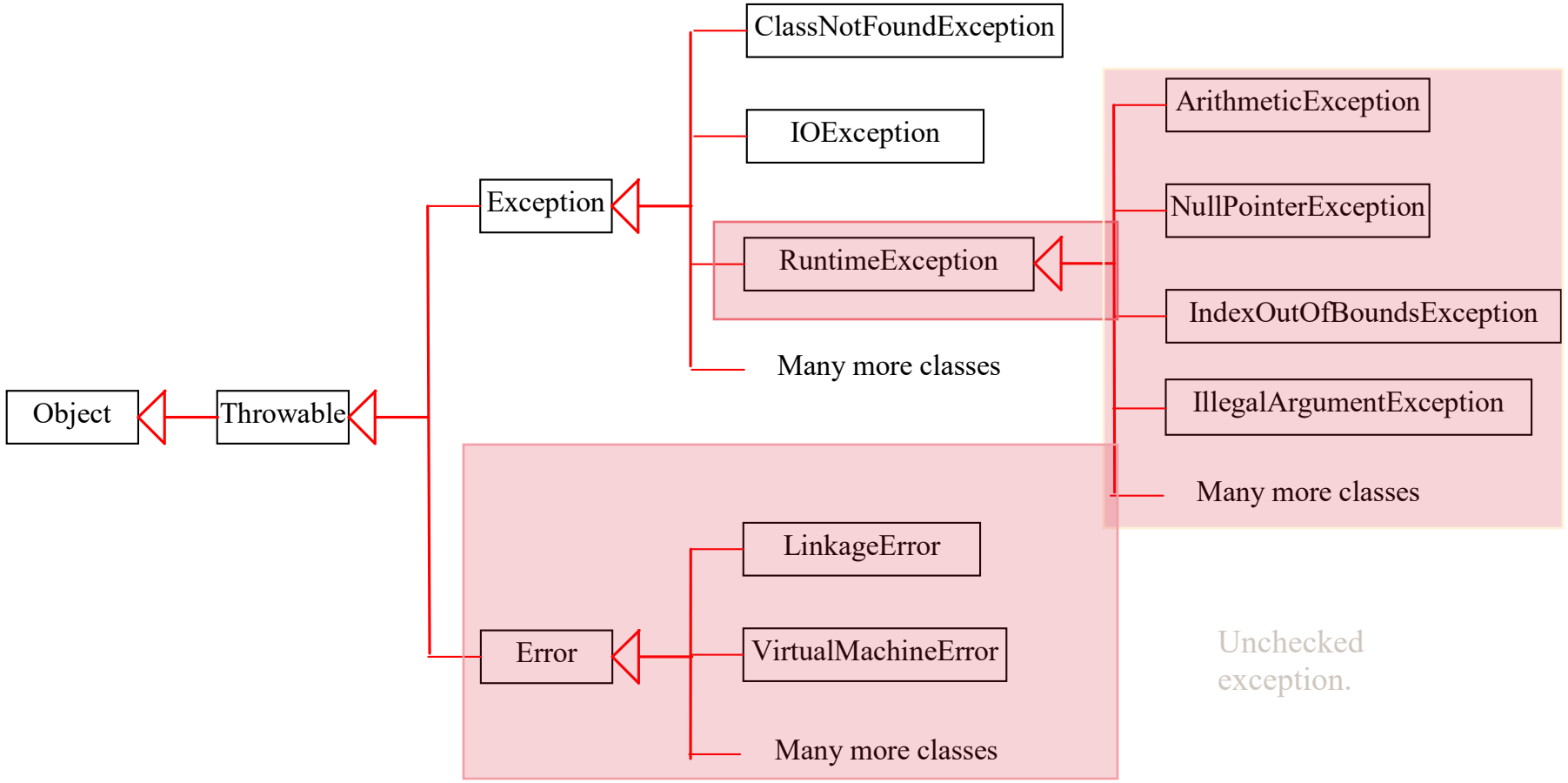
CHECKED EXCEPTIONS VS. UNCHECKED EXCEPTIONS

RuntimeException, Error and their subclasses are known as *unchecked exceptions*. All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.

UNCHECKED EXCEPTIONS

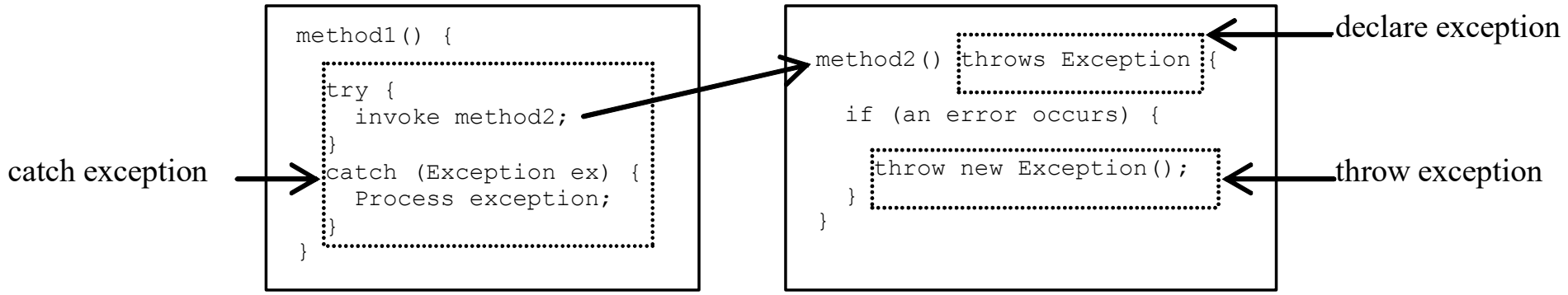
In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a NullPointerException is thrown if you access an object through a reference variable before an object is assigned to it; an IndexOutOfBoundsException is thrown if you access an element in an array outside the bounds of the array. These are the logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.

UNCHECKED EXCEPTIONS



Unchecked exception.

DECLARING, THROWING, AND CATCHING EXCEPTIONS



DECLARING EXCEPTIONS

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()
   throws IOException

public void myMethod()
   throws IOException, OtherException
```

THROWING EXCEPTIONS

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

```
TheException ex = new TheException();
throw ex;
```

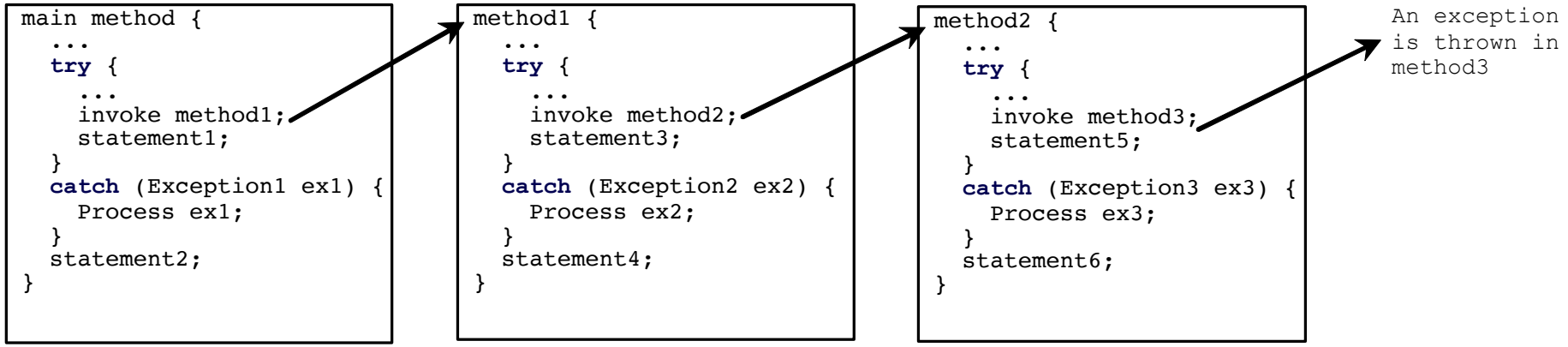
THROWING EXCEPTIONS EXAMPLE

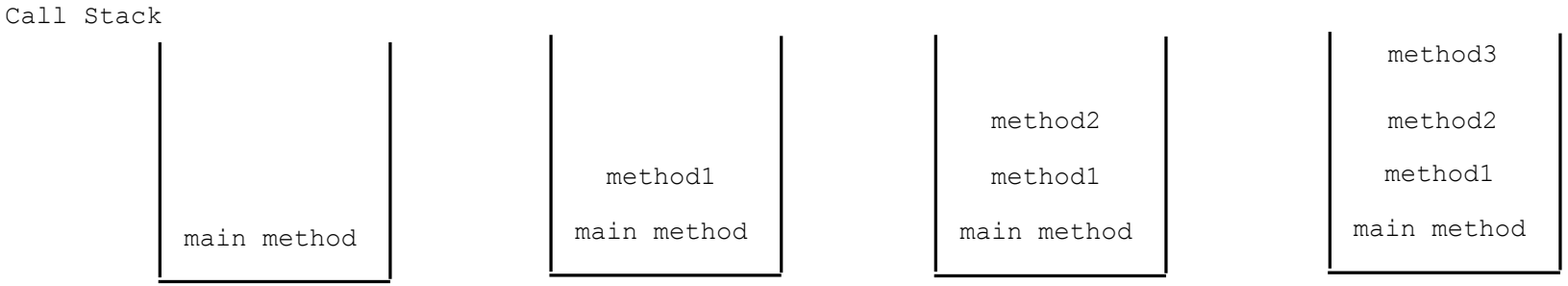
```
  /** Set a new radius */
public void setRadius(double newRadius)
    throws IllegalArgumentException {
  if (newRadius >= 0)
    radius =  newRadius;
  else
    throw new IllegalArgumentException(
      "Radius cannot be negative");
}
```

CATCHING EXCEPTIONS

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```

CATCHING EXCEPTIONS





ADDING EXCEPTION HANDLING FOR CIRCLE

Using our Circle class from last week, modified

HOW DO WE TEST (UNIT TESTING)?

- We've been writing JUnit tests
- No main() required
  - You can write one for basic "smoke tests" but really not needed since we're verifying the UUT with a complete test class
- Put on your "tester/QA" hat and try to "break" the system
- The examples in class are NOT exhaustive. Some things to consider are →

Examples

- Verify "good state" initialization after constructor
- `toString()` override returns proper value in proper format
- Values and calculations correct?
- Values and calculations correct and MUTATES instance variables properly?
- Boundary Conditions: Any constraints on values? Does the method work correctly? (e.g. with 0? Negative numbers?)
- Verify proper throw exceptions for error conditions when checked runtime errors arise? (e.g. `ArithmeticException` or `IllegalStateException` for our `SimpleFraction` class – reciprocal())
- Verify all transformations work properly (e.g. reciprocal())
- Verify any calculations on external objects/values work (e.g. .multiply(otherFraction)
- Etc.

WORKSHOP: SIMPLE FRACTIONS TESTING

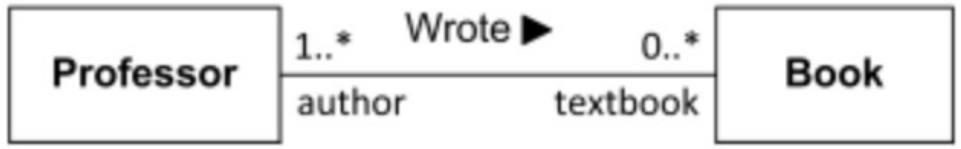
- Create a SimpleFraction class that has:
  - A constructor that takes a numerator & denominator (both integers).
    - If the denominator is negative, throw an `IllegalArgument` exception
  - "getters" for the numerator and denominator
  - A makeReciprocal method that "flips" the fraction
    - If the operation would cause a divide-by-zero error, throw an `ArithmeticExecption`
  - A toString method that represents the SimpleFraction as a string in the form: numerator / denominator

REFACTORING, REDESIGN & UML

- The Circle we worked on today has two instance variables: x and y (or something similar) to represent the center point
  - What are your thoughts about this?
  - Is there another, perhaps more coherent way to represent this information?
  - How can we refactor the code to represent this differently
- Circle "has-a" relationship with center point. This is a structural relationship, not behavior. We can capture this type of relationship with a UML "association" in a class diagram
  - UML has concepts of aggregation & composition, which are stronger

UML CLASS DIAGRAMS – RELATIONSHIPS (HAS-A)

- Association



- Aggregation & Composition

Images from https://www.uml-diagrams.org/association.html

CLASSES & ASSOCIATIONS

- How would we represent the work we did in UML today?
- Class diagram for Circle, Color, Point?

Q & A

THANKS!

- Stay safe, be encouraged, & see you next week!