CS 5004 – Object Oriented Design and Analysis

Spring 2024

Additional Reading Notes

Lecture 2: Code Style and Documentation. Object Orientation. Methods for Simple Classes. Exceptions.

Therapon Skoteiniotis, Amit Shesh, Tamara Bonaci, Abi Evans

Overview

- Commenting and documentation
  - Javadoc documentation
  - Class UML diagrams
- Writing classes under object-oriented programming paradigm
- Dealing with exceptions in Java

Relevant Reading Material

- Java Getting Started
  (https://docs.oracle.com/javase/tutorial/getStarted/index.html)
- Object-Oriented Programming Concepts
  (https://docs.oracle.com/javase/tutorial/java/concepts/index.html)
- Language Basics
  (https://docs.oracle.com/javase/tutorial/java/nutsandbolts/index.html)
  Javadoc
  (http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html)
- Unit Testing with JUnit - Tutorial
  (http://www.vogella.com/tutorials/JUnit/article.html)
- Classes (https://docs.oracle.com/javase/tutorial/java/javaOO/classes.html)
  Objects (https://docs.oracle.com/javase/tutorial/java/javaOO/objects.html)
  https://docs.oracle.com/javase/tutorial/java/javaOO/more.html
  Numbers and Strings
  (https://docs.oracle.com/javase/tutorial/java/data/index.html) Packages
  (https://docs.oracle.com/javase/tutorial/java/package/index.html)

1. Commenting and Documentation

In the last lecture, we introduced classes as entities that combine data and operations on that data. We talked about the difference between classes and objects, and we showed how to use JUnit framework to test our Java code.

In this lecture, we continue our conversation about Java and about Object Oriented programming paradigm. We will first talk about conventions to comment and document our Java code.

As you already know, it is a really good idea to write comments explaining your design and purpose. This allows you and anybody else using your code to understand what it is doing, how to use it and how it has been designed and implemented. When documenting our Java code, it is a good idea to abide by the following conventions:

- Above the class definition, explain in 1-2 sentences what this class represents. This explanation should include both **semantic details** (e.g., what the class represents from the problem statement) and **technical details** (e.g., useful for a fellow designer/programmer). Also mention any details that a user of this class may need to know to use it appropriately.
- Before each method (including the constructor and getters) write the purpose statement for the method. Also include a list of any arguments along with what they represent, and what the method returns.
- Within the method body mention any details that you think are relevant to what that method is doing.

A good rule of thumb is to assume that the audience for your comments is not you, but other designers/programmers who will use your code. If the language is such that only you can understand it fully (because you are the one who implemented it) revise the comments.

1.1. Javadoc Documentation in Java

As software developers, we are both producers and consumers of documentation. Any good code base comes with helpful documentation. For example, "official" Java documentation is available online. You can read about the String class that we used above here (https://docs.oracle.com/javase/8/docs/api/java/lang/String.html).

Java comes with a built-in tool that helps you generate such html web pages from documentation written in your Java code. This tool is called **Javadoc**. This tool "reads" your comments and converts them into html. You can use specific formatting commands within your comments to facilitate generating pleasing documentation. For example, here is the Person class with Javadoc-style comments.

```
/**

   * This class represents a person The person has a first name, last name
and an year of birth
   */
  class Person {
    private String firstName;
    private String lastName;
    private int yearOfBirth;

    /**
     * Constructs a Person object and initializes it
     * to the given first name, last name and year of birth
     * @param firstName the first name of this person
     * @param lastName the last name of this person
     * @param yearOfBirth the year of birth of this person
     */
    public Person(String firstName, String lastName, int yearOfBirth) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.yearOfBirth = yearOfBirth;
     }

    /**
     * Get the first name of this person
     * @return the first name of this person
     */
    public String getFirstName() {
      return this.firstName;
     }

    /**
     * Return the last name of this person
     * @return the last name of this person
     */
    public String getLastName() {
      return this.lastName;
     }

    /**
     * Return the year of birth of this person
     * @return the year of birth of this person
     */
    public int getYearOfBirth() {
      return this.yearOfBirth;
     }
}
```

Some conventions about Javadoc comments:

- All comments beginning with /** are Javadoc-style comments. These are the only comments that the Javadoc tool will read.
- Every line of such comments begins with * followed by a space.
- `@param` denotes information about a method argument.

- `@return` denotes information about whatever the method returns.
- Note again that since everything is within Java comments, this documentation does not affect your Java source code in any way.

1.2. Graphical Representation of Code and UML Diagrams

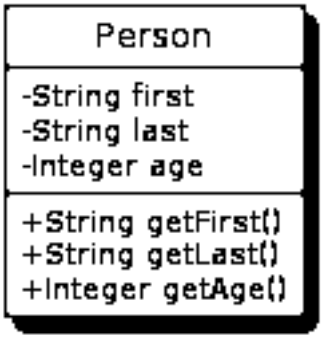
As part of designing our data for a given problem, we will often draw diagrams. The graphical notation that we will use is based on **UML** (http://www.uml.org/).

UML stands for **Unified Modeling Language**, and it is defined as a language used to model software solutions, application structures, system behavior and business processes. Two main categories of UML diagrams are:

- **Structure diagrams**
- **Behavior diagrams**

In this course, we will mostly use structure diagrams, and predominantly **class diagrams**.

As an example, let's recall the class Person from the last lecture. A **class diagram** for class Person is presented as follows:

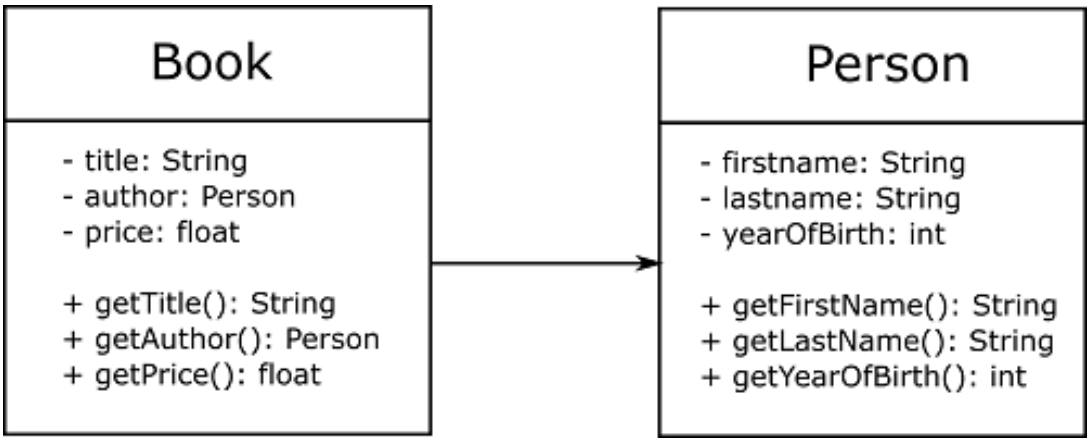


To denote a class, we use a rectangular shape with three sections, each section separated by a horizontal line:

- The first section holds the class name - Person.
- The second section lists all fields and their types
  - We use - for private fields
  - We use + for public fields
- The third section lists all the methods and their **signatures**.
  - We again use - for private methods
  - We again use + for public methods

Class diagram also allow us to represent relationships between classes. For example, in the last lecture, we considered an example of a class Book that contained an object of

class Author. Using class diagram, we can represent such a composition (has a) relationship as follows:



2. Writing Methods Under Object Oriented Programming Paradigm

2.1. Signature and purpose

Every method definition consists of the following parts:

- **A purpose statement** (which will become a part of our class Javadoc documentation)
- The type of the value returned from the method, known as the **return type**
- The **method name**, where the standard naming convention starts with a lowercase letter and uses "camelCase" to distinguish words within the name
- A **parenthesized argument list**, consisting of the type and name of each argument, separated by commas
- **The method body**, surrounded by braces; this is the code to execute when the method is invoked

Example 1: An object in String form
It is often handy to have a `String` representation of an object. This string may, for example, be used to print the contents of the object in a simple way. We will write a method called `toString`() for this purpose.

- **Method signature**
  - How should we define method `toString`()?
  - What should its signature be?

We know it needs information about the fields of the class, but nothing else. For the Person class ,we need the String to contain only the first and the last name as a single sentence (e.g. "Tom Cruise"). So, it needs to know the first name and last name, but

nothing else. **Recall the change in metaphor for Java methods: we are asking the object to operate on itself, not a function that is external to the data.**

Thus, we mean to say "Person, convert yourself into a string and return it". Since every method inside an object has access to this (the object used to call the method), we have all the data for the person that we need. Lastly this method should return the string. So, its signature will look like this:

```
/**
   Returns a string representation of this person with first
   and last name
   @return a formatted string
   */
  public String toString() {
    ...
  }
```

- **Method Body**

Once the preparatory work is done, defining the method is fairly straightforward. We simply create a string from parts and return it.

```
/**
   Returns a string representation of this person with first
   and last name
   @return a formatted string
   */
  public String toString() {
    return "" + firstName  + " " + lastName;

}
```

- **Testing**

How do we test this method? We must create a person with a specific first and last name, call its toString method and compare the String returned by it with the expected string.

The following JUnit test class shows such a test:

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;
/**
 * JUnit test class for books
 */
public class BookTest {
```

```
    private Person pat;
    private Book beaches;
    @Before
    public void setUp() {
      pat = new Person("Pat","Conroy", 1948);
      // example of books
      beaches = new Book("Beaches", this.pat, 20);
   }

    @Test
    public void testPersonString() {
      assertEquals("Pat Conroy",john.toString());
    }

}
```

Example 2: `toString()` for the Book class

We now want the `toString()` method of the Book class to return a string that contains its title, author (first and last name) and the price, one on each line. Since the price must be a monetary amount, we want it to be formatted accordingly (two decimal places).

- **Method signature**

The signature of the `toString()` method will remain the same as that of the Person class (why?).

- **Method body**

Method implementation has three parts to it:

- The title is simple: it is already a string.
- The author must be available as a single "first-name last-name". Although we can create such a string using the Person object's getter methods, we can reuse the `toString()` method we wrote for the Person class since it returns exactly the string we want!
- Getting the price is simple, but we must format it correctly.
- We must determine a way to put the above three pieces of information in separate lines. The method can be implemented as follows:

```
public String toString() {

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

```
}
```

The String class has a method format that takes a string that explains the required format. This string has "placeholders" that are replaced by the actual parameters passed to it in order.

In the above example, we required the price (a decimal number) to be formatted such that there should be exactly two numbers after the decimal point. The way to say this is "%.2f". %f normally means "placeholder for a decimal number", and "%.2f" means "placeholder for a decimal number that should be formatted with 2 numbers after the decimal point".

Notice also how this method is called by using the name of the class String instead of a String object. This is a static method, and we will see this in more detail later.

A Java String can be created by adding two strings using the + operator, just like numbers.

When we want a line break in a string, we insert the character "\n" at the appropriate place.

Here is how we can test this method:

```
  @Test
  public void testBookString() {
    String expected;
    expected = "Title: Beaches\nAuthor: Pat Conroy\n"
            +  "Price: 20.00";
    assertEquals(expected,beaches.toString());

}
```

The method signature public String toString() is chosen deliberately, because it has a special meaning in Java. Every Java class is guaranteed to have a method with this exact signature in it, and so we have just "redefined" it above for the Person and Book classes. Whenever Java needs to convert an object to a String, it uses its (guaranteed to be available) toString method. You can see this in action: in the method body of the toString for the Book class above, change this.author.toString() to simply this.author. It is surprising that although this.author is a Person object, Java is OK with adding it to a string. This is because it is using its toString method automatically to convert!

Example 3: the discounted price of a book

As is often the case, a book is offered at a discount, expressed as a percentage of the price of the book. We must now write a method `salePrice`() that computes and returns the discounted price of a book.

- **Method signature**
- How should we define the method `salePrice`?
- What should its signature be?

We know it needs a discount rate, and it should operate on a book. Recall the change in metaphor for Java methods: **we are asking the object to operate on itself, not a function that is external to the data. Thus we mean to say "Book, compute your discounted price and return it".** Since every method inside an object has access to this (the object used to call the method), we have all the data for the book that we need. Lastly this method should return the discounted price. Thus our signature will look like this:

```
  /**
     * Compute and return the price of this book with the given
discount (as a
     * percentage)
     *
     * @param discount the percentage discount to be applied
     * @return the discounted price of this book
     */
    public float salePrice(float discount) {
      ...
}
```

- **Method Body**

```
  /**
     * Compute and return the price of this book with the given
discount (as a percentage)
     *
     * @param discount the percentage discount to be applied
     * @return the discounted price of this book
     */
    public float salePrice(float discount) {
      return this.price - (this.price * discount) / 100;
}
```

- **Testing**
- How do we test this method?

We must create a book with a specific price, then call its `salePrice` method passing it a specific discount, and then verify that the discounted price returned by it matches our expected discounted price.
The following JUnit test class shows such a test:

```
  import org.junit.Before;
  import org.junit.Test;
  import static org.junit.Assert.*;
  /**
   * JUnit test class for books
   */
  public class BookTest {
```

```
    private Person pat;
    private Book beaches;
    @Before
    public void setUp() {
      pat = new Person("Pat","Conroy", 1948);
      // example of books
      beaches = new Book("Beaches", this.pat, 20);
}
    @Test
    public void testDiscount() {
      float discountedPrice = beaches.salePrice(20);
      assertEquals(16.0f,discountedPrice,0.01);
    }
}
```

In this test class, we first create the object for the author, and then use it to create the `Book` object. In the `testDiscount` test method we store the result returned by the `salePrice` method for a 20% discount and verify that, for the original price of 20, the discounted price is 16. This verification is done by the `assertEquals` method. Interestingly this method takes 3 parameters. This is because it is unwise to compare two decimal numbers directly due to precision errors (15.9999999 is not equal to 16). So, this version checks if the expected value (first parameter) is equal to the actual value (second parameter) within the error threshold passed as the third parameter.

Example 4: Methods with objects as arguments

Given two books, we would like to determine if they have the same authors. What does "same author" mean? For our purpose, two authors are the same if they have the same name and the same year of birth.

- **Method signature**

We can envision a method `sameAuthor`. Since we are going to compare two books, we need two book objects. Should they be arguments to this method? Let us briefly imagine how we would like to use this method:

```
Book book1 = new Book(...);
Book book2 = new Book(...);
//verify if book1 and book2 have the same authors
//option 1:
book1.sameAuthor(book1,book2);
```

It may seem tempting to infer that since `sameAuthor` must work with two Book objects, it must have two Book objects as arguments. However, since we are writing this method in the Book class itself, we need to call it using a Book object. It seems odd that book1 shows up twice on that line. Recall that the method will have access to this, the object used to call that method. So, we it only the other Book object, not both.

```
Book book1 = new Book(...);
Book book2 = new Book(...);
//verify if book1 and book2 have the same authors
//option 2:
book1.sameAuthor(book2);
```

We can read this as "book1, check if you have the same author as book2". Notice again how we have made this verification an operation of the book, not an external function on two books.

So, finally the signature of this method can be:

```
/**
    * check if this book has the same author as another
    * and return true if so, false otherwise
    * @param other the other book
    * @return true if the two books have the same author, false
otherwise
    */
   public boolean sameAuthor(Book other) {
```

- **Method body**

We need only the authors of the two books to complete this method. Recall what it means for two authors to be "the same": their names and their years of birth should be the same.

We may implement this method as follows: get the two Person objects for the two authors, and then compare their first and last names and their years of birth through the getter methods, all in this `sameAuthor` method.

While it may work, is it appropriate?

When we re-read the above paragraph, we notice that comparing two authors requires data only from the respective Person objects. Thus, this comparison can be performed fully within the Person class itself, as a method! The `sameAuthor` method in the Book class can then simply use it as "author 1, check if you are the same as author 2" without worrying about the details of that "sameness". In other words, it can delegate computing sameness to the Person class.

```
//Person.java
/**
 * check if this person is the same
 * as the person in the argument.
 * two persons are the same iff they
 * have the same first and last names
 * and the same years of birth
 * @param other the other person to be compared to
```

```
     * @return true if this person is the same as other, false
otherwise
     */
    public boolean same(Person other) {
      return  this.firstName.equals(other.firstName)
              && this.lastName.equals(other.lastName)
              && this.yearOfBirth == other.yearOfBirth;

}

//Book.java

   /**
     * check if this book has the same author as another
     * and return true if so, false otherwise
     * @param other the other book
     * @return true if the two books have the same author, false
otherwise
     */
    public boolean sameAuthor(Book other) {
  /* TEMPLATE:
    fields:
    this.title: String
    this.price: float
    this.author: Person
    methods:
    salePrice(float): float
    fields of parameters:
    methods of parameters:
    other.getTitle(): String
    other.getAuthor(): Person
    other.getPrice(): float
    other.salePrice(float): float
.

*//

//get the two authors and delegate

      return this.author.same(other.author);
    }
```

Two aspects that may be new:

- The String class offers a method equals that allows us to compare two String objects. Read about it in the Java 8 documentation
- The && are the logical AND operators.

- **Testing**

We wrote two new methods: so, we must test both of them accordingly.

- `same(Person other)`: This method can return one of two possible answers: true or false. There are several ways in which two authors may not be the same: different first names, different last names and/or different years of birth. Therefore, we need several cases to test for correctness

```
//PersonTest.java
  @Test
  public void testSamePerson() {
    Person benlerner = new Person("Ben","Lerner",1982);
    Person benaffleck = new Person("Ben","Affleck",1982);
    Person timlerner = new Person("Tim","Lerner",1982);
    Person anotherbenlerner = new Person("Ben","Lerner",1983);
    Person identicaltwin = new Person("Ben","Lerner",1982);
    assertFalse(benlerner.same(benaffleck));
    assertFalse(benlerner.same(timlerner));
    assertFalse(benlerner.same(anotherbenlerner));
    assertTrue(benlerner.same(identicaltwin));

}
```

- sameAuthor(Book other): This method can return one of two possible answers: true or false. As above, there are several ways in which two books may not have the same authors. But all of these cases have been tested above already! Thus, if we can assume that the above test passes, we can write a simple test for this method.

```
  @Test
  public void testSameAuthors() {
    Person anotherauthor = new Person("Pat", "Conroy II",
1948);
    Book pirateBeaches = new Book("Beaches",
            anotherauthor, 1948);
    Book sequel = new Book("Some more beaches",this.pat,1952);
    assertTrue(beaches.sameAuthor(sequel));
    assertFalse(beaches.sameAuthor(pirateBeaches));

  }
```

Example 5: Methods that return objects

Instead of merely computing the discounted price, suppose we wanted the result as an actual Book object whose details are the same as the original book, except its price is the

discounted price. This situation may arise if we are keeping track of the books we sell, and therefore need records of actual books.

- **Method signature**

We can envision a method `discountBook`. This method, as with `salePrice` would need the percentage discount. However, we want it to return a Book object instead of just the discounted price. Thus. its method signature would be:

```
 /**
  * Compute the sale price of this Book given using
  * the given discount rate (as a percentage) and
  * return a version of this book with the discounted price
  * @param discount the percentage discount to be applied to this book
  * @return the new book that is identical to this book except the price is
  * discounted
  */

//

  public Book discountBook(float discount) {
   ...
}
```

- **Method body**

We already know how to compute the discounted price. We must then create and return a new Book object with the same title and author as this one, but with the discounted price.

```
  /**
    * Compute the sale price of this Book given using
    * the given discount rate (as a percentage) and
    * return a version of this book with the discounted price
    * @param discount the percentage discount to be applied to
this book
    * @return the new book that is identical to this book except
the price is
    * discounted
    */

//

   public Book discountBook(float discount) {
   /* TEMPLATE:
   fields:
   this.title: String
```

```
    this.author: Person
    this.price: float
    constructor:
    Book(String,Person,float)
    methods:
    this.salePrice(float): float
    */
      float discountedPrice = this.salePrice(discount);
      return new Book(this.title, this.author, discountedPrice);
    }
```

Notice that we used the method `salePrice` to calculate the discounted price, instead of replicating the math. While replicating would not have been difficult, it is unadvisable. Whenever possible avoid replication of code (replicate code and thou shall replicate errors!).

- **Testing**
- How do we test the `discountBook` method?

We would have to verify that the book returned by this method has the same title and author as the original book, but the discounted price.

```
  @Test
    public void testDiscountBook() {
      Book discountedBook = beaches.discountBook(20);
      //verify that both books have the same author
      assertTrue(beaches.sameAuthor(discountedBook));
      //verify that both books have the same title

assertEquals(beaches.getTitle(),discountedBook.getTitle());
      //verify the discounted price of the new book
      assertEquals(16,discountedBook.getPrice(),0.01);

}
```

Note how `assertEquals` is used to compare titles. Titles are String, so how does `assertEquals` compare them? It uses the `equals` method of the String class that we used before. The `equals` method is special: each Java class has guaranteed to have one (just like `toString` above). Thus, `assertEquals` is capable of working with any Java objects: whether it will do what is expected depends on what the relevant equals method does.

How des Java pull off this magic of guaranteeing that every class has methods called equals and toString? We shall see later.

3. Dealing with Exceptions in Java

Let us look at the `salePrice(float)` method again. What would happen if we used it as follows:

```
Book beaches = new Book(...);
float discountPrice = beaches.salePrice(-10);
```

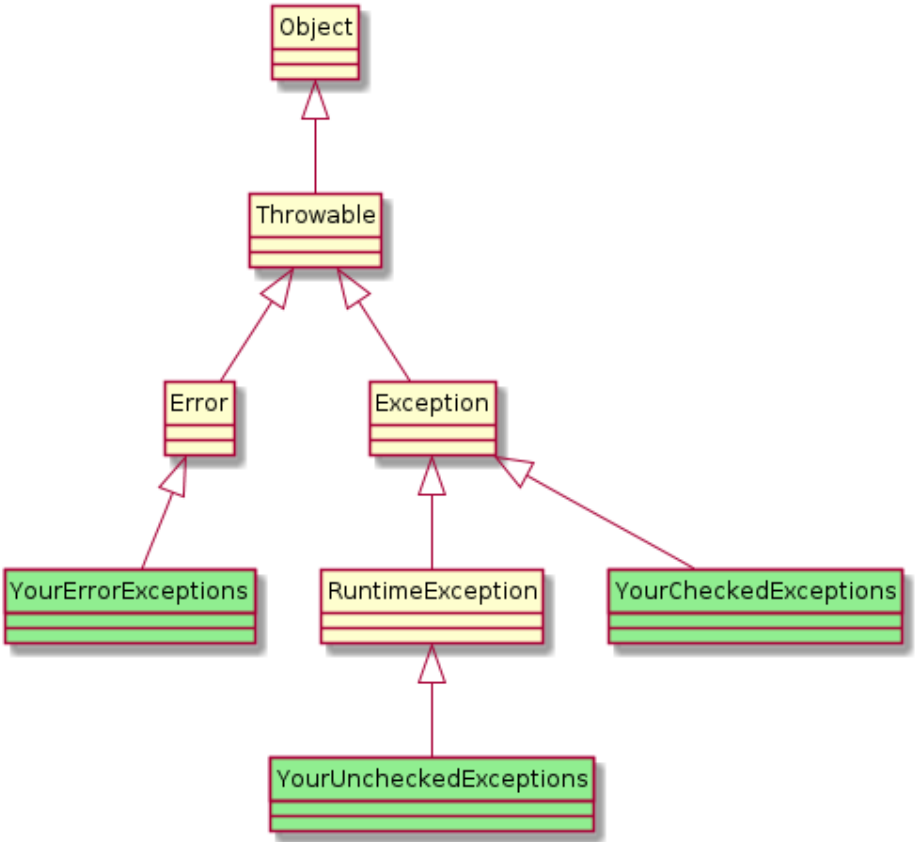
Looking at our math in this method, it would return us a negative discounted price. However. a negative percentage discount does not make sense in the given context. Java was not able to catch this error because -10 is a valid number. Ideally this method should inform its caller "thou shall not pass me a negative number for the discount", instead of using the number and returning an answer that is invalid. **Exceptions allow us to do that.**

An exception is an event that occurs during execution of a program, and disrupts the normal flow of execution. An exception occurs when something unexpected happens, whether it be invalid input, an operation that cannot be completed (e.g. the square root of a negative number) or even something that is beyond our control (e.g. attempting to read from a file that no longer exists). **Exceptions offer us a dignified way of aborting a method and sending a message to its caller that something went wrong.**

There exist three kinds of exceptions:

1. **Checked Exceptions –** capture events that a well-written application **must** anticipate, and recover from.
    a. opening a file using the file name provided by the user and cannot find that file
    b. opening a connection to a server whose address was provided to the program by an external entity
2. **Errors –** capture exceptional conditions external to the application that the application cannot anticipate or recover from.
    a. error reading from the disk
    b. error reading from the connection to the database
3. **Unchecked (Runtime) Exception –** captures events that are exceptional and internal to the application. The application typically cannot anticipate them or recover from
    a. passing in the incorrect values for the arguments to a function logical errors
    b. going over the length of a list

Java's exception hierarchy dictates the kind of exception (checked, runtime or error).



Java provides special syntax to mark code that could throw an exception as well as catching an exception. Let's look at some examples.

3.1. Writing a method with exceptions

In the `salePrice` method, an exception should occur if a negative number is passed as the percentage discount. We can change the method to the following:

```
  /**
     * Compute and return the price of this book with the given
discount (as a percentage)
     *
     * @param discount the percentage discount to be applied
     * @return the discounted price of this book
     * @throws IllegalArgumentException if a negative discount
is passed as an
     * argument
     */
    public float salePrice(float discount) throws
IllegalArgumentException {
```

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

- Java has many kinds of exceptions. Since our problem here is an invalid argument, we use the IllegalArgumentException.
- The method signature explicitly declares that it may throw an IllegalArgumentException. A method can throw multiple types of exceptions, declared in its signature separated by commas.
- The Javadoc-style comments document this possibility.
  Before using the parameter discount we check if it is negative and if so, we throw an exception. This involves creating an IllegalArgumentException with a helpful message in it and throwing it.

This method now works as follows:

- If the argument `discount` is a positive number, the method does not throw an exception and returns the discounted price, as before.
- If the argument `discount` is a negative number, the method would abort on the line throw new IllegalArgumentException(...);. It will not return anything.

3.2. Calling methods that may throw exceptions

Let us test this method, specifically by passing it a negative discount. Whenever a method is called that may throw one or more exceptions, we can enclose it in a try-catch block as follows:

```
  try {
    book.salePrice(-10);
    }
    catch (IllegalArgumentException e) {
      //This will be executed only if an
IllegalArgumentException is thrown by the above method call

}
```

Thus we try to call such a method, and if an exception is thrown, we catch it. If no exception is thrown then the catch block is ignored.

3.3. Testing methods with exceptions

We can test whether a method throws exceptions when expected using the try-catch blocks as above:

```
  @Test
  public void testIllegalDiscount() {
    float discountedPrice;
    try {
      discountedPrice = beaches.salePrice(20);
      assertEquals(16.0f,discountedPrice,0.01);
    }
    catch (IllegalArgumentException e) {
      fail("An exception should not have been thrown");
    }
    try {
      discountedPrice = beaches.salePrice(-20);
      fail("An exception should have been thrown");
    }
    catch (IllegalArgumentException e) {

} }
```

We expect the first method call

```
beaches.salePrice(20);
```

to work correctly. If an exception is thrown, the catch block will fail this test case. If the method returns a value without throwing an exception then the following `assertEquals(16.0f,discountedPrice,0.01);`

checks if the discounted price is correct (as before) and the catch block is ignored.

We expect the second method call

```
beaches.salePrice(-20);
```

to throw an exception. If an exception was indeed thrown, then the try block will abort and the statement

```
fail("An exception should have been thrown");
```

will not be executed. Catching the (expected) exception allows this test to pass. If an exception was not thrown then the

```
fail("An exception should have been thrown");
```

fails the test case.

JUnit allows us to test for exceptions in a more concise way. We must divide the above test case into two.

```
@Test
  public void testDiscount() {
    float discountedPrice = beaches.salePrice(20);
    assertEquals(16.0f, discountedPrice, 0.01);
  }
  @Test(expected = IllegalArgumentException.class)
  public void testIllegalDiscount() {
    float discountedPrice;
    discountedPrice = beaches.salePrice(-20);
}
```

The first test is as before, checking if the `salePrice` method returns the correct discounted price, if a valid discount is passed to it.

The second test has this annotation:
`@Test(expected = IllegalArgumentException.class)`.

This declares that this test expects the `IllegalArgumentException` to be thrown, and thus will abort and pass when it encounters this exception for the first time during its execution. If this method ends without an `IllegalArgumentException` thrown even once, this test fails.

3.4. Using Exceptions in General

Java has several inbuilt exception types, which are all classes. Exceptions provide us with a way to handle errors. Follow this procedure when you write a new method:

- Write the purpose statement
- Write a method signature
- Write the method body so that it works correctly under ideal circumstances.
- Carefully think about all situations that can occur, other than ideal circumstances. This includes possibly invalid inputs, invalid computations or results. These are possible errors.
- For each error:
    - If there is a way to prevent the error from happening, do it. This is the best kind of error handling. Else go to the next step.
    - If there is a way to recover from the error in this method itself, do it. There is no need to use exceptions as the recovery happens in the same method as the error. Else go to the next step.
    - Choose an appropriate exception type for the error. Declare that this method throws this exception in its signature, and throw this exception appropriately.