Object-Oriented Programming and Design with Java

HaQT

HUS
VNU UNIVERSITY OF SCIENCE

Table of Contents

Part I Java Basics

Part II Object-Oriented Programming

Part III Collections Framework

Part IV Correctness, Robustness, Efficiency

1 OOP Concepts

The OOP model is based on several concepts. These concepts must be familiar to any developer who is planning to design and program applications relying on objects. Therefore, let's start by enumerating them as follows:

- Object, Class
- Abstraction, Encapsulation, Inheritance, Polymorphism
- Association, Aggregation, Composition

1.1 What is an object?

An object is a real-world entity, such as a car, table, or cat. During its life cycle, an object has state and behaviors. For example, a cat's state can be color, name, and breed, while its behaviors can be playing, eating, sleeping, and meowing. In Java, an object is an instance of a class usually built via the new keyword, and it has state stored in fields and exposes its behavior through methods. Each instance takes some space in memory and can communicate with other objects. For example, a boy, which is another object, can caress a cat and it sleeps.

The key points:

- An object is a real-world entity.
- An object has state (fields) and behaviors (methods).
- An object represents an instance of a class.
- An object takes up some space in memory.
- An object can communicate with other objects.

1.2 What is a class?

A class is a set of instructions that are required to build a specific type of object. We can think of a class as a template, a blueprint, or a recipe that tells us how to create objects of that class. Creating an object of that class is a process called instantiation and is usually done via the new keyword. We can instantiate as many

objects as we wish. A class definition doesn't consume memory being saved as a file on the hard drive. One of the best practices that a class should follow is the Single Responsibility Principle (SRP). While conforming to this principle, a class should be designed and written to do one, and only one, thing.

The key points:

- A class is a template or a blueprint for creating objects.
- A class doesn't consume memory.
- A class can be instantiated multiple times.
- A class does one, and only one, thing.

1.3 What is abstraction?

Abstraction is one of the main OOP concepts that strive to make things as simple as possible for the user. In other words, abstraction exposes the user only to the things that are relevant to them and hides the remainder of the details. In OOP terms, we say that an object should expose to its users only a set of high-level operations, while the internal implementation of those operations is hidden.

Abstraction allows the user to focus on what the application does instead of how it does it. This way, abstraction reduces the complexity of exposing the things, increases code reusability, avoids code duplications, and sustains low coupling and high cohesion. Moreover, it maintains the security and discretion of the application by exposing only the important details.

In Java, abstraction can be achieved via abstract classes and interfaces.

The key points:

- Abstraction is the concept of exposing to the user only those things that are relevant to them and hiding the remainder of the details.
- Abstraction allows the user to focus on what the application does instead of how it does it.
- Abstraction is achieved in Java via abstract classes and interfaces.

Example. Let's consider a real-life example: a man driving a car. The man knows what each pedal does and what the steering wheel does, but he doesn't know how these things are done internally by the car. He doesn't know about the inner mechanisms that empower these things. This is what abstraction is. So, we said that a man is driving a car. The man can speed up or slow down the car via the

corresponding pedals. He also can turn left and right with the aid of the steering wheel. All these actions are grouped in an interface named **Car**:

```
public interface Car {
    public void speedUp();
    public void slowDown();
    public void turnRight();
    public void turnLeft();
    public String getCarType();
}
```

Next, each type of car should implement the **Car** interface and override these methods to provide the implementation of these actions. This implementation is hidden from the user (the man driving the car). For example, the **ElectricCar** class appears as follows (in reality, in place of *System.out.println*, we have complex business logic):

```
public class ElectricCar implements Car {
    private final String carType;

    public ElectricCar(String carType) {
        this.carType = carType;
    }

    @Override
    public void speedUp() {
        System.out.println("Speed up the electric car");
    }

    @Override
    public void slowDown() {
        System.out.println("Slow down the electric car");
    }

    @Override
    public void turnRight() {
        System.out.println("Turn right the electric car");
    }

    @Override
    public void turnLeft() {
        System.out.println("Turn left the electric car");
    }

```

```
    @Override
29  public String getCarType() {
      return this.carType;
31  }
}
```

The user of this class has access to these public methods without being aware of the implementation:

```
public class TestCar {
2   public static void main(String[] args) {
      Car electricCar = new ElectricCar("BMW");
4     System.out.println("Driving the electric car: " + electricCar.getCarType());
      electricCar.speedUp();
6     electricCar.turnLeft();
      electricCar.slowDown();
8   }
}
```

The output is listed as follows:

Command window

```
1 Driving the electric car: BMW
  Speed up the electric car
3 Turn left the electric car
  Slow down the electric car
```

1.4 What is encapsulation?

Encapsulation binds together the code and data in a single unit of work (a class) and acts as a defensive shield that doesn't allow external code to access this data directly.

Mainly, it is the technique of hiding the object state from the outer world and exposing a set of public methods for accessing this state. When each object keeps its state private inside a class, we can say that encapsulation was achieved. This is why encapsulation is also referenced as the **data-hiding** mechanism. The code that takes advantage of encapsulation is loosely coupled (for example, we can change

the names of the class variables without breaking the client code), reusable, secure (the client is not aware of how data is manipulated inside the class), and easy to test (it is easier to test methods than fields).

In Java, encapsulation can be achieved via the access modifiers, *public*, *private*, and *protected*. Commonly, when an object manages its own state, its state is declared via private variables and is accessed and/or modified via public methods.

Let's consider an example: a **Cat** class can have its state represented by fields such as mood, hungry, and energy. While the code external to the **Cat** class cannot modify any of these fields directly, it can call public methods, such as $play()$, $feed()$, and $sleep()$ that modify the **Cat** state internally. The **Cat** class may also have private methods that are not accessible outside the class, such as $meow()$. This is encapsulation.

The key points:

- Encapsulation is the technique whereby the object state is hidden from the outer world and a set of public methods for accessing this state are exposed.
- Encapsulation is achieved when each object keeps its state private, inside a class.
- Encapsulation is known as the data-hiding mechanism.
- Encapsulation has a number of important advantages associated with it, such as loosely coupled, reusable, secure, and easy-to-test code.
- In Java, encapsulation is implemented via the access modifiers – public, private, and protected.

Example. The **Cat** class from our example can be coded as indicated in the following code block. Notice that the state of this class was encapsulated via private fields, and is therefore not directly accessible from outside the class:

```
public class Cat {
  private int mood;
  private int hungry;
  private int energy;

  public void sleep() {
    System.out.println("Sleep ... ");
    energy++;
    hungry++;
  }

  public void play() {
    System.out.println("Play ... ");
```

```
14      mood++;
        energy--;
16      meow();
    }
18
    public void feed() {
20      System.out.println("Feed ... ");
        hungry--;
22      mood++;
        meow();
24  }

26  private void meow(){
        System.out.println("Meow!");
28  }

30  public int getMood() {
        return mood;
32  }

34  public int getHungry() {
        return hungry;
36  }

38  public int getEnergy() {
        return energy;
40  }
}
```

The only way to modify the state is via the public methods, $play()$, $feed()$, and $sleep()$, as in the following example:

```
1 public class TestCat {
    public static void main(String[] args) {
3       Cat cat = new Cat();
        cat.feed();
5       cat.play();
        cat.feed();
7       cat.sleep();
        System.out.println("Energy: " + cat.getEnergy());
9       System.out.println("Mood: " + cat.getMood());
        System.out.println("Hungry: " + cat.getHungry());
11  }
}
```

1.5 What is inheritance?

Inheritance is one of the core concepts of OOP. It allows an object to be based on another object, which is useful when different objects are pretty similar and share some common logic, but they are not identical. Inheritance sustains code reusability by allowing an object to reuse the code of another object while it adds its own logic as well.

In order to achieve inheritance, we reuse the common logic and extract the unique logic in another class. This is known as an IS-A relationship, also referenced as a parent-child relationship. It is just like saying Foo IS-A Buzz type of thing. For example, cat IS-A feline, and train IS-A vehicle. An IS-A relationship is the unit of work used to define hierarchies of classes.

In Java, inheritance is accomplished via the *extends* keyword by deriving the child from its parent. The child can reuse the fields and methods of its parent and add its own fields and methods. The inherited object is referenced as the superclass, or the parent class, and the object that inherits the superclass is referenced as the subclass, or the child class.

In Java, inheritance cannot be multiple; therefore, a subclass or child class cannot inherit fields and methods of more than one superclass or parent class.

For example, an **Employee** class (parent class) can define the common logic of any employee in a software company, while another class (child class), named **Programmer**, can extend the **Employee** to use this common logic and add logic specific to a programmer. Other classes can extend the **Programmer** or **Employee** classes as well.

The key points:

- Inheritance allows an object to be based on another object.
- Inheritance sustains code reusability by allowing an object to reuse the code of another object and adds its own logic as well.
- Inheritance is known as an IS-A relationship, also referenced as a parent-child relationship.
- In Java, inheritance is achieved via the extends keyword.
- The inherited object is referenced as the superclass, and the object that inherits the superclass is referenced as the subclass.
- In Java, multiple classes cannot be inherited.

Example. The **Employee** class is quite simple. It wraps the id and name of the employee:

```java
public class Employee {
  private String id;
  private String name;

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

  public String getId() {
    return this.id;
  }

  public String getName() {
    return this.name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

Then, the **Programmer** class extends the **Employee**. As any employee, a programmer has an id and a name, but they are also assigned to a team:

```java
public class Programmer extends Employee {
  private String team;

  public Programmer(String id, String name, String team) {
    super(id, name);
    this.team = team;
  }

  public String getTeam() {
    return this.team;
  }

  public void setTeam(String team) {
    this.team = team;
  }
}
```

Let's test inheritance by creating a **Programmer** and calling $getName()$, inherited from the **Employee** class, and $getTeam()$, inherited from the **Programmer** class:

```
public class TestInheritance {
  public static void main(String[] args) {
    Programmer programmer = new Programmer("Joana Nimar", "Toronto");
    String name = programmer.getName();
    String team = programmer.getTeam();
    System.out.println(name + " is assigned to the " + team + " team");
  }
}
```

1.6 What is polymorphism?

Polymorphism is a word composed of two Greek words: poly, which means many, and morph, which means forms. Therefore, polymorphism means many forms. In the OOP context, polymorphism allows an object to behave differently in certain cases or, in other words, allows an action to be accomplished in different ways (approaches).

One way to implement polymorphism is via method overloading. This is known as compile-time polymorphism because the compiler can identify at compile time which form of an overloaded method to call (multiple methods with the same name but different arguments). So, depending on which form of the overloaded method is called, the object behaves differently. For example, a class named **Triangle** can define multiple methods named $draw()$ with different arguments.

Another way to implement polymorphism is via method overriding, and this is the common approach when we have an IS-A relationship. It is known as run-time polymorphism, or Dynamic Method Dispatch. Typically, we start with an interface containing a bunch of methods. Next, each class implements this interface and overrides these methods to provide a specific behavior. This time, polymorphism allows us to use any of these classes exactly like its parent (the interface) without any confusion of their types. This is possible because, at runtime, Java can distinguish between these classes and knows which one is used. For example, an interface named **Shape** can declare a method named $draw()$, and the **Triangle**, **Rectangle**, and **Circle** classes implement the **Shape** interface and override the $draw()$ method to draw the corresponding shape.

The key points:

- Polymorphism means many forms in Greek.
- Polymorphism allows an object to behave differently in certain cases.
- Polymorphism can be shaped via method overloading (known as Compile-Time

Polymorphism) or via method overriding in the case of an IS-A relationship (known as Runtime Polymorphism).

Polymorphism via method overloading (compile time)

The **Triangle** class contains three $draws()$ statics methods, as follows:

```
public class Triangle {
  public static void draw() {
    System.out.println("Draw default triangle ...");
  }

  public static void draw(String color) {
    System.out.println("Draw a triangle of color " + color);
  }

  public static void draw(int size, String color) {
    System.out.println("Draw a triangle of color " + color
      + " and scale it up with the new size of " + size);
  }
}
```

Next, notice how the corresponding $draw()$ method is called:

```
Triangle triangle = new Triangle();
triangle.draw();
triangle.draw("red");
triangle.draw(10, "blue");
```

Polymorphism via method overriding (runtime)

This time, the $draw()$ method is declared in an interface, as follows:

```
public interface Shape {
  public void draw();
}
```

The **Triangle**, **Rectangle**, and **Circle** classes implement the **Shape** interface and override the $draw()$ method to draw the corresponding shape:

```java
public class Triangle implements Shape {
  @Override
  public void draw() {
    System.out.println("Draw a triangle ...");
  }
}
```

```java
public class Rectangle implements Shape {
  @Override
  public void draw() {
    System.out.println("Draw a rectangle ...");
  }
}
```

```java
public class Circle implements Shape {
  @Override
  public void draw() {
    System.out.println("Draw a circle ...");
  }
}
```

Next, we create a **Triangle**, a **Rectangle**, and a **Circle**. For each of these instances, let's call the $draw()$ method:

```java
public class TestOverriding {
  public static void main(String[] args) {
    Shape triangle = new Triangle();
    triangle.draw();

    Shape rectangle = new Rectangle();
    rectangle.draw();

    Shape circle = new Circle();
    circle.draw();
  }
}
```

1.7 What is association?

Association is one of the core concepts of OOP. The association goal is to define the relation between two classes independent of one another and is also referenced as the multiplicity relation between objects. There is no owner of the association. The objects involved in an association can use one another (bidirectional association), or only one uses the other one (unidirectional association), but they have their own life span. Association can be unidirectional/bidirectional, one-to-one, one-to-many, many-to-one, and many-to-many.

For example, between the **Person** and **Address** objects, we may have a bidirectional many-to-many relationship. In other words, a person can be associated with multiple addresses, while an address can belong to multiple people. However, people can exist without addresses, and vice versa.

The key points:

- Association defines the relation between two classes that are independent of one another.
- Association has no owner.
- Association can be one-to-one, one-to-many, many-to-one, and many-to-many.

Example. The **Person** and **Address** classes are as follows:

```java
public class Person {
  private String id;
  private String name;

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

  public String getId() {
    return this.id;
  }

  public String getName() {
    return this.name;
  }

  public void setName(String name) {
    this.name = name;
  }
}
```

```
public class Address {
  private String city;
  private String zip;

  public Address(String city, String zip) {
    this.city = city;
    this.zip = zip;
  }

  public String getCity() {
    return this.city;
  }

  public void setCity(String city) {
    this.city = city;
  }

  public String getZip() {
    return this.zip;
  }

  public void setZip(String zip) {
    this.zip = zip;
  }
}
```

The association between **Person** and **Address** is accomplished in the $main()$ method, as shown in the following code block:

```
public class TestAssociation {
  public static void main(String[] args) {
    Person person1 = new Person("Andrei");
    Person person2 = new Person("Marin");

    Address address1 = new Address("Banesti", "107050");
    Address address2 = new Address("Bucuresti", "229344");
    // Association between classes in the main method

    System.out.println(person1.getName() + " lives at address "
      + address2.getCity() + ", " + address2.getZip()
      + " but it also has an address at "
      + address1.getCity() + ", " + address1.getZip());

    System.out.println(person2.getName() + " lives at address "
      + address1.getCity() + ", " + address1.getZip()
```

```
        + " but it also has an address at "
        + address2.getCity() + ", " + address2.getZip());
    }
}
```

1.8 What is aggregation?

Mainly, aggregation is a special case of unidirectional association. While an association defines the relationship between two classes independent of one another, aggregation represents a HAS-A relationship between these two classes. In other words, two aggregated objects have their own life cycle, but one of the objects is the owner of the HAS-A relationship. Having their own life cycle means that ending one object will not affect the other object.

For example, a **TennisPlayer** has a **Racket**. This is a unidirectional association since a Racket cannot have a **TennisPlayer**. Even if the **TennisPlayer** dies, the **Racket** is not affected.

The key points:

- Aggregation is a special case of unidirectional association.
- Aggregation represents a HAS-A relationship.
- Two aggregated objects have their own life cycle, but one of the objects is the owner of the HAS-A relationship.

Example. We start with the **Rocket** class. This is a simple representation of a tennis racket:

```
public class Racket {
    private String type;
    private int size;
    private int weight;

    public Racket(String type, int size, int weight) {
        this.type = type;
        this.size = size;
        this.weight = weight;
    }

    public String getType() {
        return this.type;
```

```
14    }

16    public void setType(String type) {
        this.type = type;
18    }

20    public int getSize() {
        return this.size;
22    }

24    public void setSize(int size) {
        this.size = size;
26    }

28    public int getWeight() {
        return this.weight;
30    }

32    public void setWeight(int weight) {
        this.weight = weight;
34    }
  }
```

A **TennisPlayer** HAS-A **Racket**. Therefore, the **TennisPlayer** class must be capable of receiving a **Racket** as follows:

```
1  public class TennisPlayer {
     private String name;
3    private Racket racket;

5    public TennisPlayer(String name, Racket racket) {
       this.name = name;
7      this.racket = racket;
     }
9
     public String getName() {
11     return this.name;
     }
13
     public void setName(String name) {
15     this.name = name;
     }
17
     public Racket getRacket() {
19     return this.racket;
     }
```

```
21
    public void setRacket(Racket racket) {
23      this.racket = racket;
    }
25 }
```

Next, we create a **Racket** and a **TennisPlayer** that uses this **Racket**:

```
1 public class TestAggregation {
    public static void main(String[] args) {
3     Racket racket = new Racket("Babolat Pure Aero", 100, 300);
      TennisPlayer player = new TennisPlayer("Rafael Nadal", racket);
5
      System.out.println("Player " + player.getName()
7       + " plays with " + player.getRacket().getType());
    }
9 }
```

1.9 What is composition?

Primarily, composition is a more restrictive case of aggregation. While aggregation represents a HAS-A relationship between two objects having their own life cycle, composition represents a HAS-A relationship that contains an object that cannot exist on its own. In order to highlight this coupling, the HAS-A relationship can be named PART-OF as well.

For example, a **Car** has an **Engine**. In other words, the engine is PART-OF the car. If the car is destroyed, then the engine is destroyed as well. Composition is said to be better than inheritance because it sustains code reuse and the visibility control of objects.

The key points:

- Composition is a more restrictive case of aggregation.
- Composition represents a HAS-A relationship that contains an object that cannot exist on its own.
- Composition sustains code reuse and the visibility control of objects.

Example. The **Engine** class is as follows:

```java
public class Engine {
  private String type;
  private int horsepower;

  public Engine(String type, int horsepower) {
    this.type = type;
    this.horsepower = horsepower;
  }

  public String getType() {
    return this.type;
  }

  public void setType(String type) {
    this.type = type;
  }

  public int getHorsepower() {
    return this.horsepower;
  }

  public void setHorsepower(int horsepower) {
    this.horsepower = horsepower;
  }
}
```

Next, we have the **Car** class. Check out the constructor of this class. Since **Engine** is part of **Car**, we create it with the **Car**:

```java
public class Car {
  private final String name;
  private final Engine engine;

  public Car(String name) {
    this.name = name;
    this.engine = new Engine("petrol", 300);
  }

  public int getHorsepower() {
    return engine.getHorsepower();
  }

  public String getName() {
    return name;
  }
```

```
17
    public void setName(String name) {
19      this.name = name;
    }
21  }
```

Next, we can test composition from the $main()$ method as follows:

```
1  public class TestComposition {
     public static void main(String[] args) {
3      Car car = new Car("MyCar");
       System.out.println("Horsepower: " + car.getHorsepower());
5    }
   }
```

2 The SOLID principles

The SOLID principles were introduced by Robert C. Martin in his 2000 paper "Design Principles and Design Patterns". These concepts were later built upon by Michael Feathers, who introduced us to the SOLID acronym. And in the last 20 years, these five principles have revolutionized the world of object-oriented programming, changing the way that we write software.

So, what is SOLID and how does it help us write better code? Simply put, Martin and Feathers' design principles encourage us to create more maintainable, understandable, and flexible software. Consequently, as our applications grow in size, we can reduce their complexity and save ourselves a lot of headaches further down the road!

By way of a quick remainder, SOLID is an acronym of the following:

- **S**: Single Responsibility Principle
- **O**: Open Closed Principle
- **L**: Liskov's Substitution Principle
- **I**: Interface Segregation Principle
- **D**: Dependency Inversion Principle

2.1 What is S?

S is the first principle from SOLID and is known as the **Single Responsibility Principle (SRP)**. This principle translates to the fact that one class should have one, and only one, responsibility.

This is a very important principle that should be followed in any type of project for any type of class (model, service, controller, manager class, and so on). As long as we write a class for only one goal, we will sustain high maintainability and visibility control across the application modules. In other words, by sustaining high maintainability, this principle has a significant business impact, and by providing visibility control across the application modules, this principle sustains encapsulation.

The key points:

- S stands for the Single Responsibility Principle (SRP).
- S stands for One class should have one, and only one, responsibility.

- S tells us to write a class for only one goal.
- S sustains high maintainability and visibility control across the application modules.

Example

You want to calculate the area of a rectangle. The dimensions of the rectangle are initially given in meters and the area is computed in meters as well, but we want to be able to convert the computed area to other units, such as inches. Let's see the approach that breaks the SRP.

Breaking the SRP

Implementing the preceding problem in a single class, **RectangleAreaCalculator**, can be done as follows. But this class does more than one thing: it breaks SRP. Keep in mind that, typically, when you use the word and to express what a class does, this is a sign that the SRP is broken. For example, the following class computes the area and converts it to inches:

```
public class RectangleAreaCalculator {
  private static final double INCH_TERM = 0.0254d;
  private final int width;
  private final int height;

  public RectangleAreaCalculator(int width, int height) {
    this.width = width;
    this.height = height;
  }

  public int area() {
    return width * height;
  }

  // This method breaks SRP
  public double metersToInches(int area) {
    return area / INCH_TERM;
  }
}
```

Since this code contravenes the SRP, we must fix it in order to follow the SRP.

Following the SRP

The situation can be remedied by removing the $metersToInches()$ method from **RectangleAreaCalculator**, as follows:

```java
public class RectangleAreaCalculator {
  private final int width;
  private final int height;

  public RectangleAreaCalculator(int width, int height) {
    this.width = width;
    this.height = height;
  }

  public int area() {
    return width * height;
  }
}
```

Now, **RectangleAreaCalculator** does only one thing (it computes the rectangle area), thereby observing the SRP.

Next, $metersToInches()$ can be extracted in a separate class. Moreover, we can add a new method for converting from meters to feet as well:

```java
public class AreaConverter {
  private static final double INCH_TERM = 0.0254d;
  private static final double FEET_TERM = 0.3048d;

  public static double metersToInches(int area) {
    return area / INCH_TERM;
  }

  public static double metersToFeet(int area) {
    return area / FEET_TERM;
  }
}
```

This class also follows the SRP.

2.2 What is O?

O is the second principle from SOLID, and is known as the **Open Closed Principle (OCP)**. This principle stands for software components should be open for extension, but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application. This means that our classes should be designed and written in such a way that other developers

can change the behavior of these classes by simply extending them. So, our classes should not contain constraints that will require other developers to modify our classes in order to accomplish their job – other developers should only extend our classes to accomplish their job.

While we must sustain software extensibility in a versatile, intuitive, and non-harmful way, we don't have to think that other developers will want to change the whole logic or the core logic of our classes. Primarily, if we follow this principle, then our code will act as a good framework that doesn't give us access to modify their core logic, but we can modify their flow and/or behavior by extending some classes, passing initialization parameters, overriding methods, passing different options, and so on.

Example

You have different shapes (for example, rectangles, circles) and we want to sum their areas. First, let's see the implementation that breaks the OCP.

Breaking the OCP

Each shape will implement the **Shape** interface. Therefore, the code is pretty straightforward:

```
public interface Shape {

}
```

```
public class Rectangle implements Shape {
  private final int width;
  private final int height;

  public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }

  public int getWidth() {
    return this.width;
  }

  public void setWidth(int width) {
    this.width = width;
```

```
    }

    public int getHeight() {
      return this.height;
    }

    public void setHeight(int height) {
      this.height = height;
    }
}
```

```
public class Circle implements Shape {
    private final int radius;

    public Circle(int radius) {
      this.radius = radius;
    }

    public int getRadius() {
      return this.radius;
    }

    public void setRadius(int radius) {
      this.radius = radius;
    }
}
```

At this point, we can easily use the constructors of these classes to create rectangles and circles of different sizes. Once we have several shapes, we want to sum their areas. For this, we can define an **AreaCalculator** class as follows:

```
public class AreaCalculator {
    private final List<Shape> shapes;

    public AreaCalculator(List<Shape> shapes) {
      this.shapes = shapes;
    }

    // Adding more shapes requires us to modify this class.
    // This code is not OCP compliant.
    public double sum() {
```

```
        int sum = 0;
        for (Shape shape : shapes) {
            if (shape instanceof Circle) {
                sum += Math.PI * Math.pow(((Circle) shape).getRadius(), 2);
            } else {
                if (shape instanceof Rectangle) {
                    sum += ((Rectangle) shape).getHeight() * ((Rectangle) shape).getWidth();
                }
            }
        }

        return sum;
    }
}
```

Since each shape has its own formula for area, we require an if-else (or switch) structure to determine the type of shape. Furthermore, if we want to add a new shape (for example, a triangle), we have to modify the **AreaCalculator** class to add a new if case. This means that the preceding code breaks the OCP. Fixing this code to observe the OCP imposes several modifications in all classes. Hence, be aware that fixing code that doesn't follow the OCP can be quite tricky, even in the case of a simple example.

Following the OCP

The main idea is to extract from **AreaCalculator** the area formula of each shape in the corresponding **Shape** class. Hence, the rectangle will compute its area, the circle as well, and so on. To enforce the fact that each shape must calculate its area, we add the $area()$ method to the **Shape** contract:

```
public interface Shape {
    public double area();
}
```

Next, **Rectangle** and **Circle** implements **Shape** as follows:

```
public class Rectangle implements Shape {
    private final int width;
    private final int height;
```

```
    public Rectangle(int width, int height) {
      this.width = width;
      this.height = height;
    }

    @Override
    public double area() {
      return width * height;
    }
  }
```

```
  public class Circle implements Shape {
    private final int radius;

    public Circle(int radius) {
      this.radius = radius;
    }

    @Override
    public double area() {
      return Math.PI * Math.pow(radius, 2);
    }
  }
```

Now, the **AreaCalculator** can loop the list of shapes and sum the areas by calling the proper $area()$ method:

```
  public class AreaCalculator {
    private final List<Shape> shapes;

    public AreaCalculator(List<Shape> shapes) {
      this.shapes = shapes;
    }

    public double sum() {
      int sum = 0;
      for (Shape shape : shapes) {
        sum += shape.area();
      }
      return sum;
    }
```

The code is OCP-compliant. We can add a new shape and there is no need to modify the **AreaCalculator**. So, **AreaCalculator** is closed for modifications and, of course, is open for extension.

2.3 What is L?

L is the third principle from SOLID and is known as **Liskov's Substitution Principle (LSP)**. This principle stands for derived types must be completely substitutable for their base types. This means that the classes that extend our classes should be usable across the application without causing failures. More precisely, this principle sustains the fact that objects of subclasses must behave in the same way as the objects of superclasses, so every subclass (or derived class) should be capable of substituting their superclass without any issues.

Most of the time, this is useful for runtime-type identification followed by the cast. For example, consider $foo(p)$, where p is of the type **T**. Then, $foo(q)$ should work fine if q is of the type **S** and **S** is a subtype of **T**.

The key points:

- L stands for Liskov's Substitution Principle (LSP).
- L stands for Derived types must be completely substitutable for their base types.
- L sustains the fact that objects of subclasses must behave in the same way as the objects of superclasses.
- L is useful for runtime-type identification followed by the cast.

Example

We have a chess club that accepts three types of members: Premium, VIP, and Free. We have an abstract class named **Member** that acts as the base class, and three subclasses – **PremiumMember**, **VipMember**, and **FreeMember**. Let's see whether each of these member types can substitute the base class.

Breaking the LSP

The **Member** class is abstract, and it represents the base class for all members of our chess club:

```
public abstract class Member {
  private final String name;

  public Member(String name) {
    this.name = name;
  }

  public abstract void joinTournament();
  public abstract void organizeTournament();
}
```

The **PremiumMember** class can join chess tournaments or organize such tournaments as well. So, its implementation is quite simple:

```
public class PremiumMember extends Member {
  public PremiumMember(String name) {
    super(name);
  }

  @Override
  public void joinTournament() {
    System.out.println("Premium member joins tournament");
  }

  @Override
  public void organizeTournament() {
    System.out.println("Premium member organize tournament");
  }
}
```

The **VipMember** class is roughly the same as **PremiumMember**, so we can skip it and focus on the **FreeMember** class. The **FreeMember** class can join tournaments, but cannot organize tournaments. This is an issue that we need to tackle in the $organizeTournament()$ method. We can throw an exception with a meaningful message or we can display a message as follows:

```
public class FreeMember extends Member {
  public FreeMember(String name) {
    super(name);
  }
```

```
5
    @Override
7   public void joinTournament() {
      System.out.println("Classic member joins tournament ...");
9   }

11  // This method breaks Liskov's Substitution Principle
    @Override
13  public void organizeTournament() {
      System.out.println("A free member cannot organize tournaments");
15  }
}
```

But throwing an exception or displaying a message doesn't mean that we follow LSP. Since a free member cannot organize tournaments, it cannot be a substitute for the base class, therefore it breaks the LSP. Check out the following list of members:

```
List<Member> members = List.of(
2   new PremiumMember("Jack Hores"),
    new VipMember("Tom Johns"),
4   new FreeMember("Martin Vilop")
);
```

The following loop reveals that our code is not LSP-compliant because when the **FreeMember** class has to substitute the **Member** class, it cannot accomplish its job since **FreeMember** cannot organize chess tournaments:

```
1 for (Member member : members) {
    member.organizeTournament();
3 }
```

This situation is a showstopper. We cannot continue the implementation of our application. We must redesign our solution to obtain a code that is LSP-compliant. So let's do this!

Following the LSP

The refactoring process starts by defining two interfaces meant to separate the two actions, joining and organizing chess tournaments:

```java
public interface TournamentJoiner {
    public void joinTournament();
}
```

```java
public interface TournamentOrganizer {
    public void organizeTournament();
}
```

Next, the abstract base class implements these two interfaces as follows:

```java
public abstract class Member implements TournamentJoiner, TournamentOrganizer {
    private final String name;

    public Member(String name) {
        this.name = name;
    }
}
```

PremiumMember and **VipMember** remain untouched. They extend the **Member** base class. However, the **FreeMember** class, which cannot organize tournaments, will not extend the **Member** base class. It will implement the **TournamentJoiner** interface only:

```java
public class FreeMember implements TournamentJoiner {
    private final String name;

    public FreeMember(String name) {
        this.name = name;
    }

    @Override
    public void joinTournament() {
        System.out.println("Free member joins tournament ...");
    }
}
```

Now, we can define a list of members who can join chess tournaments as follows:

```
List<TournamentJoiner> members = List.of(
  new PremiumMember("Jack Hores"),
  new VipMember("Tom Johns"),
  new FreeMember("Martin Vilop")
);
```

Looping this list and substituting the **TournamentJoiner** interface with each type of member works as expected and observes the LSP:

```
// this code respects LSP
for (TournamentJoiner member : members) {
  member.joinTournament();
}
```

Following the same logic, a list of members who can organize chess tournaments can be written as follows:

```
List<TournamentOrganizer> members = List.of(
  new PremiumMember("Jack Hores"),
  new VipMember("Tom Johns")
);
```

FreeMember doesn't implement the **TournamentOrganizer** interface. Therefore, it cannot be added to this list. Looping this list and substituting the **TournamentOrganizer** interface with each type of member works as expected and follows the LSP:

```
// This code respects LSP
for (TournamentOrganizer member : members) {
  member.organizeTournament();
}
```

Done! Now we have an LSP-compliant code.

2.4 What is I?

It is the fourth principle from SOLID, and is known as the **Interface Segregation Principle (ISP)**. This principle stands for clients should not be forced to implement unnecessary methods that they will not use. In other words, we should split an interface into two or more interfaces until clients are not forced to implement methods that they will not use. By doing so, we can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

The key points:

- I stands for the Interface Segregation Principle (ISP).
- I stands for clients should not be forced to implement unnecessary methods that they will not use.
- I splits an interface into two or more interfaces until clients are not forced to implement methods that they will not use.
- We can ensure that implementing classes only need to be concerned about the methods that are of interest to them.

Example

Let's consider the **Connection** interface, which has three methods: $connect()$, $socket()$, and $http()$. A client may want to implement this interface only for connections via HTTP. Therefore, they don't need the $socket()$ method. Most of the time, the client will leave this method empty, and this is a bad design. In order to avoid such situations, simply split the **Connection** interface into two interfaces; **SocketConnection** with the $socket()$ method, and **HttpConnection** with the $http()$ method. Both interfaces will extend the **Connection** interface that remains with the common method, $connect()$.

Breaking the ISP

The **Connection** interface defines three methods as follows:

```
public interface Connection {
    public void socket();
    public void http();
    public void connect();
}
```

WwwPingConnection is a class that pings different websites via HTTP; hence, it requires the *http*() method, but doesn't need the *socket*() method. Notice the dummy *socket*() implementation – since **WwwPingConnection** implements **Connection**, it is forced to provide an implementation to the *socket*() method as well:

```
public class WwwPingConnection implements Connection {
    private final String www;

    public WwwPingConnection(String www) {
        this.www = www;
    }

    @Override
    public void http() {
        System.out.println("Setup an HTTP connection to " + www);
    }

    @Override
    public void connect() {
        System.out.println("Connect to " + www);
    }

    // This method breaks Interface Segregation Principle
    @Override
    public void socket() {
    }
}
```

Having an empty implementation or throwing a meaningful exception from methods that are not needed, such as *socket*(), is a really ugly solution. Check the following code:

```
WwwPingConnection www = new WwwPingConnection 'www.yahoo.com');
www.socket();   // We can call this method!
www.connect();
```

What do we expect to obtain from this code? A working code that does nothing, or an exception caused by the *connect*() method because there is no HTTP endpoint? Or, we can throw an exception from *socket*() of the type: **Socket** is not supported!. Then, why is it here?! Hence, it is now time to refactor the code to follow the ISP.

Following the ISP

In order to comply with the ISP, we need to segregate the **Connection** interface. Since the *connect*() method is required by any client, we leave it in this interface:

```
1 public interface Connection {
    public void connect();
3 }
```

The *http*() and *socket*() methods are distributed in to separate interfaces that extend the **Connection** interface as follows:

```
1 public interface HttpConnection extends Connection {
    public void http();
3 }
```

```
1 public interface SocketConnection extends Connection {
    public void socket();
3 }
```

This time, the **WwwPingConnection** class can implement only the **HttpConnection** interface and use the *http*() method:

```
1 public class WwwPingConnection implements HttpConnection {
    private final String www;
3
    public WwwPingConnection(String www) {
5     this.www = www;
    }
7
    @Override
9   public void http() {
      System.out.println("Setup an HTTP connection to " + www);
11  }

13  @Override
```

```
    public void connect() {
15      System.out.println("Connect to " + www);
    }
17  }
```

Done! Now, the code follows the ISP.

2.5 What is D?

D is the last principle from SOLID and is known as the **Dependency Inversion Principle (DIP)**. This principle stands for depend on abstractions, not on concretions. This means that we should rely on abstract layers to bind concrete modules together instead of having concrete modules that depend on other concrete modules. To accomplish this, all concrete modules should expose abstractions only. This way, the concrete modules allow extension of the functionality or plug-in in another concrete module while retaining the decoupling of concrete modules. Commonly, high coupling occurs between high-level concrete modules and low-level concrete modules.

Example

A database JDBC URL, **PostgreSQLJdbcUrl** , can be a low-level module, while a class that connects to the database may represent a high-level module, such as $ConnectToDatabase\#connect()$.

Breaking the DIP

If we pass to the $connect()$ method an argument of the **PostgreSQLJdbcUrl** type, then we have violated the DIP. Let's look at the code of **PostgreSQLJdbcUrl** and **ConnectToDatabase**:

```
1   public class PostgreSQLJdbcUrl {
      private final String dbName;
3
      public PostgreSQLJdbcUrl(String dbName) {
5       this.dbName = dbName;
      }
7
      public String get() {
9       return "jdbc://   ...   " + this.dbName;
```

```
    }
}
```

```
public class ConnectToDatabase {
    public void connect(PostgreSQLJdbcUrl postgresql) {
        System.out.println("Connecting to " + postgresql.get());
    }
}
```

If we create another type of JDBC URL (for example, **MySQLJdbcUrl**), then we cannot use the preceding *connect*(PostgreSQLJdbcUrl *postgreSQL*) method. So, we have to drop this dependency on concrete and create a dependency on abstraction.

Following the DIP

The abstraction can be represented by an interface that should be implemented by each type of JDBC URL:

```
public interface JdbcUrl {
    public String get();
}
```

Next, **PostgreSQLJdbcUrl** implements **JdbcUrl** to return a JDBC URL specific to **PostgreSQL** databases:

```
public class PostgreSQLJdbcUrl implements JdbcUrl {
    private final String dbName;

    public PostgreSQLJdbcUrl(String dbName) {
        this.dbName = dbName;
    }

    @Override
    public String get() {
        return "jdbc://   ...   " + this.dbName;
    }
```

```
}
```

In precisely the same manner, we can write **MySQLJdbcUrl**, **OracleJdbcUrl**, and so on. Finally, the $ConnectToDatabase\#connect()$ method is dependent on the **JdbcUrl** abstraction, so it can connect to any JDBC URL that implements this abstraction:

```
public class ConnectToDatabase {
  public void connect(JdbcUrl jdbcUrl) {
    System.out.println("Connecting to " + jdbcUrl.get());
  }
}
```

Done!

3 Exception Handling by Examples

3.1 Introduction

An exception is an *abnormal event* that arises during the execution of the program and disrupts the normal flow of the program. Abnormality do occur when your program is running. For example, you might expect the user to enter an integer, but receive a text string; or an unexpected I/O error pops up at runtime. What really matters is "what happens after an abnormality occurred?" In other words, "how the abnormal situations are handled by your program." If these exceptions are not handled properly, the program terminates abruptly and may cause severe consequences. For example, the network connections, database connections and files may remain opened; database and file records may be left in an inconsistent state.

Java has a built-in mechanism for handling runtime errors, referred to as *exception handling*. This is to ensure that you can write robust programs for mission-critical applications.

Older programming languages such as C have some drawbacks in exception handing. For example, suppose the programmer wishes to open a file for processing:

1. The programmers are not made to aware of the exceptional conditions. For example, the file to be opened may not necessarily exist. The programmer therefore did not write codes to test whether the file exists before opening the file.

2. Suppose the programmer is aware of the exceptional conditions, he/she might decide to finish the main logic first, and write the exception handling codes later – this "later", unfortunately, usually never happens. In other words, you are not force to write the exception handling codes together with the main logic.

3. Suppose the programmer decided to write the exception handling codes, the exception handling codes intertwine with the main logic in many if-else statements. This makes main logic hard to follow and the entire program hard to read. For example,

```
1 if ( file  exists ) {
    open  file ;
```

```
while (there is more records to be processed) {
  if (no IO errors) {
    process the file record
  } else {
    handle the errors
  }
}
if (file is opened) close the file;
} else {
report the file does not exist;
}
```

Java overcomes these drawbacks by building the exception handling into the language rather than leaving it to the discretion of the programmers:

1. You will be informed of the exceptional conditions that may arise in calling a method - Exceptions are declared in the method's signature.
2. You are forced to handle exceptions while writing the main logic and cannot leave them as an afterthought - Your program cannot compiled without the exception handling codes.
3. Exception handling codes are separated from the main logic - via the try-catch-finally construct.

Let's look into these three points in more details.

Point 1: Exceptions must be declared

As an example, suppose that you want to use a **java.util.Scanner** to perform formatted input from a disk file. The signature of the **Scanner**'s constructor with a **File** argument is given as follows:

```
public Scanner(File source) throws FileNotFoundException;
```

The method's signature informs the programmers that an exceptional condition "file not found" may arise. By declaring the exceptions in the method's signature, programmers are made to aware of the exceptional conditions in using the method.

Point 2: Exceptions must be handled

If a method declares an exception in its signature, you cannot use this method without handling the exception - you can't compile the program.

Example 1. The program did not handle the exception declared, resutled in compilation error.

```
1 import java.util.Scanner;
  import java.io.File;
3
  public class ScannerFromFile {
5   public static void main(String[] args) {
      Scanner in = new Scanner(new File("test.in"));
7     // do something ...
    }
9 }
```

Command window

```
1 ScannerFromFile.java:5: unreported exception java.io.FileNotFoundException; must be
      caught or declared to be thrown
  Scanner in = new Scanner(new File("test.in"));
3 ^
```

To use a method that declares an exception in its signature, you MUST either:

1. provide exception handling codes in a "try-catch" or "try-catch-finally" construct, or
2. not handling the exception in the current method, but declare the exception to be thrown up the call stack for the next higher-level method to handle.

Example 2. Catch the exception via a "try-catch" (or "try-catch-finally") construct.

```
1 import java.util.Scanner;
  import java.io.File;
3 import java.io.FileNotFoundException;

5 public class ScannerFromFileWithCatch {
    public static void main(String[] args) {
7     try {
        Scanner in = new Scanner(new File("test.in"));
9       // do something if no exception ...
        // you main logic here in the try-block
11    } catch (FileNotFoundException ex) {   // error handling separated from
        // the main logic
```

```
            ex.printStackTrace();              // print the stack trace
        }
    }
}
```

If the file cannot be found, the exception is caught in the catch-block. In this example, the error handler simply prints the stack trace, which provides useful information for debugging. In some situations, you may need to perform some clean-up operations, or open another file instead. Take note that the main logic in the try-block is separated from the error handling codes in the catch-block.

Example 3. You decided not to handle the exception in the current method, but throw the exception up the call stack for the next higher-level method to handle.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class ScannerFromFileWithThrow {
   public static void main(String[] args) throws FileNotFoundException {
      // To be handled by next higher-level method
      Scanner in = new Scanner(new File("test.in"));
      // This method may throw FileNotFoundException
      // Main logic here ...
   }
}
```

In this example, you decided not to handle the **FileNotFoundException** thrown by the $Scanner(File)$ method (with try-catch). Instead, the caller of $Scanner(File)$ - the $main()$ method - declares in its signature "throws **FileNotFoundException**", which means that this exception will be thrown up the call stack, for the next higher-level method to handle. In this case, the next higher-level method of $main()$ is the JVM, which simply terminates the program and prints the stack trace.

Point 3: Main logic is separated from the exception handling codes

As shown in **Example 2**, the main logic is contained in the try-block, while the exception handling codes are kept in the catch-block(s) separated from the main logic. This greatly improves the readability of the program.

For example, a Java program for file processing could be as follows:

```
try {
  // Main logic here
  open file;
  process file;
    ......
} catch (FileNotFoundException ex) {    // Exception handlers below
  // Exception handler for "file not found"
} catch (IOException ex) {
  // Exception handler for "IO errors"
} finally {
  close file;        // always try to close the file
}
```

3.2 Method Call Stack

A typical application involves many levels of method calls, which is managed by a so-called method call stack. A stack is a last-in-first-out queue. In the following example, $main()$ method invokes $methodA()$; $methodA()$ calls $methodB()$; $methodB()$ calls $methodC()$.

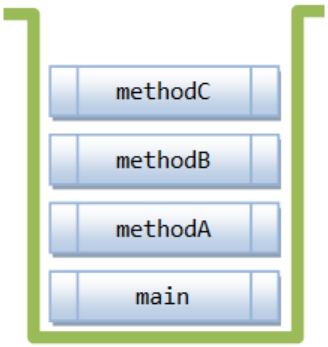
```
public class MethodCallStackDemo {
  public static void main(String[] args) {
    System.out.println("Enter main()");
    methodA();
    System.out.println("Exit main()");
  }

  public static void methodA() {
    System.out.println("Enter methodA()");
    methodB();
    System.out.println("Exit methodA()");
  }

  public static void methodB() {
    System.out.println("Enter methodB()");
    methodC();
    System.out.println("Exit methodB()");
  }

  public static void methodC() {
    System.out.println("Enter methodC()");
    System.out.println("Exit methodC()");
  }
}
```

Command window

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exit methodC()
Exit methodB()
Exit methodA()
Exit main()
```

As seen from the output, the sequence of events is:

1. JVM invoke the $main()$.
2. $main()$ pushed onto call stack, before invoking $methodA()$.
3. $methodA()$ pushed onto call stack, before invoking $methodB()$.
4. $methodB()$ pushed onto call stack, before invoking $methodC()$.
5. $methodC()$ completes.
6. $methodB()$ popped out from call stack and completes.



Method Call Stack
(Last-in-First-out Queue)

7. $methodA()$ popped out from the call stack and completes.
8. $main()$ popped out from the call stack and completes. Program exits.

Suppose that we modify $methodC()$ to carry out a "divide-by-0" operation, which triggers a **ArithmeticException**:

```
public static void methodC() {
   System.out.println("Enter methodC()");
   System.out.println(1 / 0);    // divide-by-0 triggers an ArithmeticException
   System.out.println("Exit methodC()");
}
```

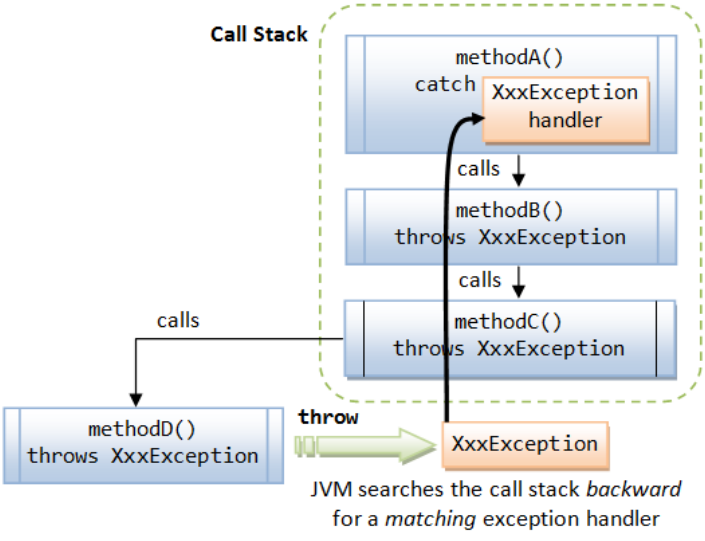
The exception message clearly shows the method call stack trace with the relevant statement line numbers:

Command window

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)
 at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16)
 at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10)
 at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

$methodC()$ triggers an **ArithmeticException**. As it does not handle this exception, it popped off from the call stack immediately. $methodB()$ also does not handle this exception and popped off the call stack. So does $methodA()$ and $main()$ method. The $main()$ method passes back to JVM, which abruptly terminates the program and print the call stack trace, as shown.

3.3 Exception & Call Stack



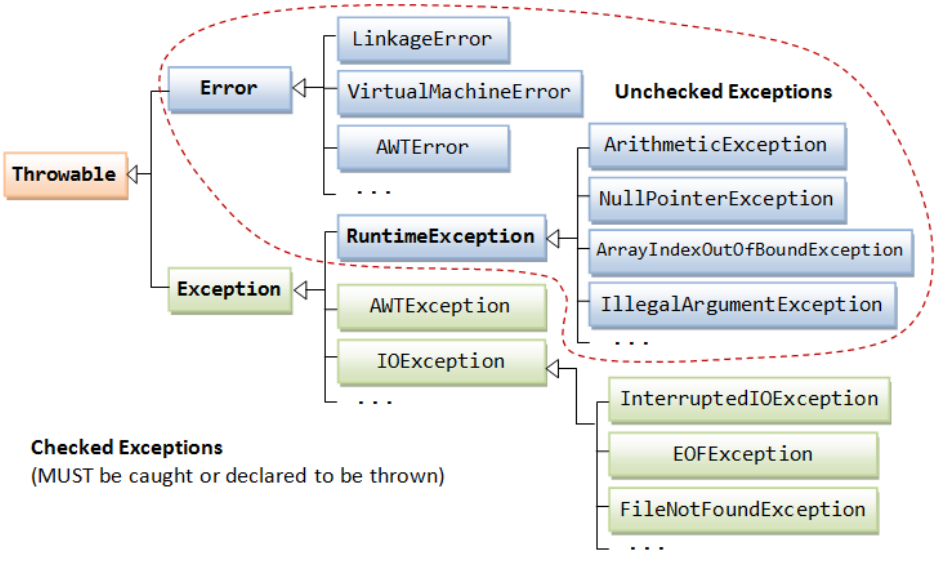
When an exception occurs inside a Java method, the method creates an **Exception** object and passes the **Exception** object to the JVM (in Java term, the method "throw" an **Exception**). The **Exception** object contains the type of the exception, and the state of the program when the exception occurs. The JVM is responsible for finding an *exception handler* to process the **Exception** object. It searches backward through the call stack until it finds a matching exception handler for that particular

class of **Exception** object (in Java term, it is called "catch" the **Exception**). If the JVM cannot find a matching exception handler in all the methods in the call stack, it terminates the program.

This process is illustrated as follows. Suppose that $methodD()$ encounters an abnormal condition and throws a **XxxException** to the JVM. The JVM searches backward through the call stack for a matching exception handler. It finds $methodA()$ having a **XxxException** handler and passes the exception object to the handler. Notice that $methodC()$ and $methodB()$ are required to declare "throws **XxxException**" in their method signatures in order to compile the program.

3.4 Exception Classes - Throwable, Error, Exception & RuntimeException

The figure below shows the hierarchy of the **Exception** classes. The base class for all **Exception** objects is **java.lang.Throwable**, together with its two subclasses **java.lang.Exception** and **java.lang.Error**.



- The **Error** class describes internal system errors (e.g., **VirtualMachineError**, **LinkageError**) that rarely occur. If such an error occurs, there is little that you can do and the program will be terminated by the Java runtime.
- The **Exception** class describes the error caused by your program (e.g. **FileNotFoundException**, **IOException**). These errors could be caught and handled by your program (e.g., perform an alternate action or do a graceful exit by closing all the files, network and database connections).

3.5 Checked vs. Unchecked Exceptions

As illustrated, the subclasses of **Error** and **RuntimeException** are known as *unchecked exceptions*. These exceptions are not checked by the compiler, and hence, need not be caught or declared to be thrown in your program. This is because there is not much you can do with these exceptions. For example, a "divide by 0" triggers an **ArithmeticException**, array index out-of-bound triggers an **ArrayIndexOutOfBoundException**, which are really programming logical errors that shall be been fixed in compiled-time, rather than leaving it to runtime exception handling.

All the other exception are called *checked exceptions*. They are checked by the compiler and must be caught or declared to be thrown.

3.6 Exception Handling Operations

Five keywords are used in exception handling: try, $catch$, $finally$, $throws$ and $throw$ (take note that there is a difference between $throw$ and $throws$).

Java's exception handling consists of three operations:

1. Declaring exceptions;
2. Throwing an exception; and
3. Catching an exception.

Declaring Exceptions

A Java method must declare in its signature the types of checked exception it may "throw" from its body, via the keyword "throws". For example, suppose that $methodD()$ is defined as follows:

```
1 public void methodD() throws XxxException, YyyException {
    // method body throw XxxException and YyyException
3 }
```

The method's signature indicates that running $methodD()$ may encounter two checked exceptions: **XxxException** and **YyyException**. In other words, some of the abnormal conditions inside $methodD()$ may trigger **XxxException** or **YyyException**.

Exceptions belonging to **Error**, **RuntimeException** and their subclasses need not be declared. These exceptions are called unchecked exceptions because they are not checked by the compiler.

Throwing an Exception

When a Java operation encounters an abnormal situation, the method containing the erroneous statement shall create an appropriate **Exception** object and throw it to the Java runtime via the statement "throw **XxxException**". For example,

```
public void methodD() throws XxxException, YyyException {   // Method's signature
   // Method's body
    ...
   // XxxException occurs
   if ( ... ) {
      throw new XxxException(...);   // Construct an XxxException object and throw to JVM
   }
    ...
   // YyyException occurs
   if ( ... ) {
      throw new YyyException(...);   // Construct an YyyException object and throw to JVM
   }
    ...
}
```

Note that the keyword to declare exception in the method's signature is "throws" and the keyword to throw an exception object within the method's body is "throw".

Catching an Exception

When a method throws an exception, the JVM searches backward through the call stack for a matching exception handler. Each exception handler can handle one particular class of exception. *An exception handler handles a specific class can also handle its subclasses.* If no exception handler is found in the call stack, the program terminates.

For example, suppose $methodD()$ declares that it may throw **XxxException** and **YyyException** in its signature, as follows:

```
public void methodD() throws XxxException, YyyException {
    ......
}
```

To use $methodD()$ in your program (says in $methodC()$), you can either:

1. Wrap the call of $methodD()$ inside a try-catch (or try-catch-finally) as follows. Each catch-block can contain an exception handler for one type of exception.

```
public void methodC() {   // No exception declared
   ......

   try {
      ......
      // Uses methodD() which declares XxxException & YyyException
      methodD();
      ......
   } catch (XxxException ex) {
      // Exception handler for XxxException
      ......
   } catch (YyyException ex) {
      // Exception handler for YyyException
      ......
   } finally {    // Optional
      // These codes always run, used for cleaning up
      ......
   }

   ......
}
```

2. Suppose that $methodC()$ who calls $methodD()$ does not wish to handle the exceptions (via a try-catch), it can declare these exceptions to be thrown up the call stack in its signature as follows:

```
public void methodC() throws XxxException, YyyException {
                              // For next higher-level method to handle
   ...
   // Uses methodD() which declares "throws XxxException, YyyException"
   methodD();    // No need for try-catch
   ...
}
```

In this case, if a **XxxException** or **YyyException** is thrown by $methodD()$, JVM will *terminate* $methodD()$ as well as $methodC()$ and pass the exception object up the call stack to the caller of $methodC()$.

3.7 try-catch-finally

The syntax of try-catch-finally is:

```
try {
   // main logic, uses methods that may throw Exceptions
     ......
} catch (Exception1 ex) {
   // error handler for Exception1
     ......
} catch (Exception2 ex) {
   // error handler for Exception1
     ......
} finally {    // finally is optional
   // clean up codes, always executed regardless of exceptions
     ......
}
```

If no exception occurs during the running of the try-block, all the catch-blocks are skipped, and finally-block will be executed after the try-block. If one of the statements in the try-block throws an exception, the Java runtime ignores the rest of the statements in the try-block, and begins searching for a matching exception handler. It matches the exception type with each of the catch-blocks sequentially. If a catch-block catches that exception class or catches a superclass of that exception, the statement in that catch-block will be executed. The statements in the finally-block are then executed after that catch-block. The program continues into the next statement after the try-catch-finally, unless it is pre-maturely terminated or branch-out.

If none of the catch-block matches, the exception will be passed up the call stack. The current method executes the finally clause (if any) and popped off the call stack. The caller follows the same procedures to handle the exception.

The finally block is almost certain to be executed, regardless of whether or not exception occurs (unless JVM encountered a severe error or a $System.exit()$ is called in the catch block).

Example 1.

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
```

```
public class TryCatchFinally {
  public static void main(String[] args) {
    try {  // Main logic
      System.out.println("Start of the main logic");
      System.out.println("Try opening a file ...");
      Scanner in = new Scanner(new File("test.in"));
      System.out.println("File Found, processing the file ...");
      System.out.println("End of the main logic");
    } catch (FileNotFoundException ex) {  // Error handling separated from the main
        ↪ logic
      System.out.println("File Not Found caught ...");
    } finally {  // Always run regardless of exception status
      System.out.println("finally-block runs regardless of the state of exception");
    }

    // After the try-catch-finally
    System.out.println("After try-catch-finally, life goes on ...");
  }
}
```

- This is the output when the **FileNotFoundException** triggered:

Command window

```
Start of the main logic
Try opening a file ...
File Not Found caught ...
finally-block runs regardless of the state of exception
After try-catch-finally, life goes on ...
```

- This is the output when no exception triggered:

Command window

```
Start of the main logic
Try opening a file ...
File Found, processing the file ...
End of the main logic
finally-block runs regardless of the state of exception
After try-catch-finally, life goes on ...
```

Example 2.

```
1  public class MethodCallStackDemo {
     public static void main(String[] args) {
3      System.out.println("Enter main()");
       methodA();
5      System.out.println("Exit main()");
     }
7
     public static void methodA() {
9      System.out.println("Enter methodA()");
       try {
11       System.out.println(1/0);
         // A divide-by-0 triggers an ArithmeticException - an unchecked exception
13       // This method does not catch ArithmeticException
         // It runs the "finally" and popped off the call stack
15     } finally {
         System.out.println("finally in methodA()");
17     }

19     System.out.println("Exit methodA()");
     }
21 }
```

Command window

```
1  Enter main()
   Enter methodA()
3  finally in methodA()
   Exception in thread "main" java.lang.ArithmeticException: / by zero
5  at MethodCallStackDemo.methodA(MethodCallStackDemo.java:11)
   at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

try-catch-finally

- A try-block must be accompanied by at least one catch-block or a finally-block.
- You can have multiple catch-blocks. Each catch-block catches only one type of exception.
- A catch block requires one argument, which is a throwable object (i.e., a subclass of **java.lang.Throwable**), as follows:

```
catch (AThrowableSubClass aThrowableObject) {
  // Exception handling codes
}
```

- You can use the following methods to retrieve the type of the exception and the state of the program from the Throwable object:
  - $printStackTrace()$: Prints this **Throwable** and its call stack trace to the standard error stream System.err. The first line of the outputs contains the result of $toString()$, and the remaining lines are the stack trace. This is the most common handler, if there is nothing better that you can do. For example,

```
try {
   Scanner in = new Scanner(new File("test.in"));
   // process the file here
     ......
} catch (FileNotFoundException ex) {
   ex.printStackTrace();
}
```

    You can also use $printStackTrace$(PrintStream s) or $printStackTrace$(PrintWriter s).
  - $getMessage()$: Returns the message specified if the object is constructed using constructor $Throwable(String\ message)$.
  - $toString()$: Returns a short description of this Throwable object, consists of the name of the class, a colon ':', and a message from $getMessage()$.
- A catch block catching a specific exception class can also catch its subclasses. Hence, $catch$(Exception ex) ... catches all kinds of exceptions. However, this is not a good practice as the exception handler that is too general may unintentionally catches some subclasses' exceptions it does not intend to.
- The order of catch-blocks is important. A subclass must be caught (and placed in front) before its superclass. Otherwise, you receive a compilation error "exception **XxxException** has already been caught".
- The finally-block is meant for cleanup code such as closing the file, database connection regardless of whether the try block succeeds. The finally block is always executed (unless the catch-block pre-maturely terminated the current method).

What if I really don't care about the exceptions

Certainly not advisable other than writing toy programs. But to bypass the compilation error messages triggered by methods declaring unchecked exceptions, you could declare "throws Exception" in your $main()$ (and other methods), as follows:

```
public static void main(String[] args) throws Exception {   // Throws all subclass
      ↪ of Exception to JRE
   Scanner in = new Scanner(new File("test.in"));   // Declares "throws
         ↪ FileNotFoundException"
      .....
   // Other exceptions
}
```

Overriding and Overloading Methods

An overriding method must have the same argument list and return-type (or subclass of its original from JDK 1.5). An overloading method must have different argument list, but it can have any return-type.

An overriding method cannot have more restricted access. For example, a method with protected access may be overridden to have protected or public access but not private or default access. This is because an overridden method is considered to be a replacement of its original, hence, it cannot be more restrictive.

An overriding method cannot declare exception types that were not declared in its original. However, it may declare exception types are the same as, or subclass of its original. It needs not declare all the exceptions as its original. It can throw fewer exceptions than the original, but not more.

An overloading method must be differentiated by its argument list. It cannot be differentiated by the return-type, the exceptions, and the modifier, which is illegal. It can have any return-type, access modifier, and exceptions, as long as it can be differentiated by the argument list.

3.8 Common Exception Classes

ArrayIndexOutOfBoundsException: thrown by JVM when your code uses an array index, which is is outside the array's bounds. For example,

```
int[] anArray = new int[3];
System.out.println(anArray[3]);
```

Command window

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 3
```

NullPointerException: thrown by the JVM when your code attempts to use a null reference where an object reference is required. For example,

```
String[] strs = new String[3];
System.out.println(strs[0].length());
```

Command window

```
Exception in thread "main" java.lang.NullPointerException
```

NumberFormatException: Thrown programmatically (e.g., by $Integer.parseInt()$) when an attempt is made to convert a string to a numeric type, but the string does not have the appropriate format. For example,

```
Integer.parseInt("abc");
```

Command window

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "abc
    "
```

ClassCastException: thrown by JVM when an attempt is made to cast an object reference fails. For example,

```
Object o = new Object();
Integer i = (Integer) o;
```

Command window

```
Exception in thread "main" java.lang.ClassCastException: java.lang.Object cannot
    be cast to java.lang.Integer
```

IllegalArgumentException: thrown programmatically to indicate that a method has been passed an illegal or inappropriate argument. You could re-use this exception for your own methods.

IllegalStateException: thrown programmatically when a method is invoked and the program is not in an appropriate state for that method to perform its task. This typically happens when a method is invoked out of sequence, or perhaps a method is only allowed to be invoked once and an attempt is made to invoke it again.

NoClassDefFoundError: thrown by the JVM or class loader when the definition of a class cannot be found. Prior to JDK 1.7, you will see this exception call stack trace if you try to run a non-existent class. JDK 1.7 simplifies the error message to "Error: Could not find or load main class xxx".

3.9 Creating Your Own Exception Classes

You should try to reuse the **Exception** classes provided in the JDK, e.g., **IndexOutOfBoundException**, **ArithmeticException**, **IOException**, and **IllegalArugmentException**. But you can always create you own **Exception** classes by extending from the class **Exception** or one of its subclasses.

Note that **RuntimeException** and its subclasses are not checked by the compiler and need not be declared in the method's signature. Therefore, use them with care, as you will not be informed and may not be aware of the exceptions that may occur by using that method (and therefore do not have the proper exception handling codes) – a bad software engineering practice.

Example

```
// Create our own exception class by subclassing Exception.
// This is a checked exception
public class MyMagicException extends Exception {
   public MyMagicException(String message) {
      super(message);
   }
}
```

```
public class TestMyMagicException {
  // This method "throw MyMagicException" in its body.
  // MyMagicException is checked and need to be declared in the method's signature
  public static void magic(int number) throws MyMagicException {
    if (number == 8) {
      throw (new MyMagicException("you hit the magic number"));
    }
    System.out.println("hello");   // skip if exception triggered
  }

  public static void main(String[] args) {
    try {
      magic(9);     // does not trigger exception
      magic(8);     // trigger exception
    } catch (MyMagicException ex){    // exception handler
      ex.printStackTrace();
    }
  }
}
```

The output is as follows:

Command window

```
hello
MyMagicException: you hit the magic number
at MyMagicExceptionTest.magic(MyMagicExceptionTest.java:6)
at MyMagicExceptionTest.main(MyMagicExceptionTest.java:14)
```

4 Exceptions

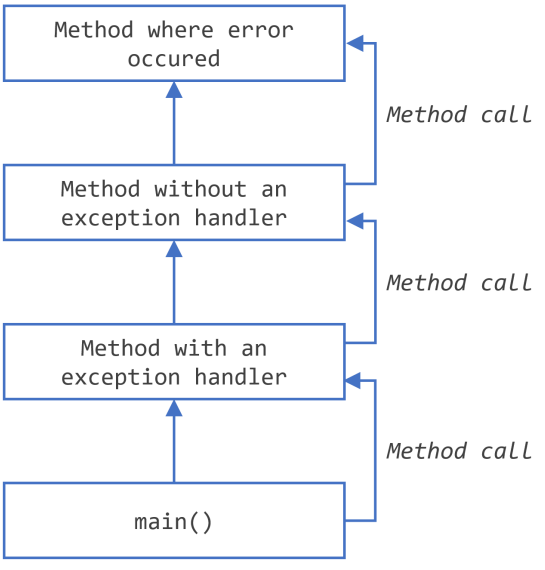
4.1 What Is an Exception?

4.1.1 Definition of Exception.

The term exception is shorthand for the phrase "exceptional event." An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The 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.

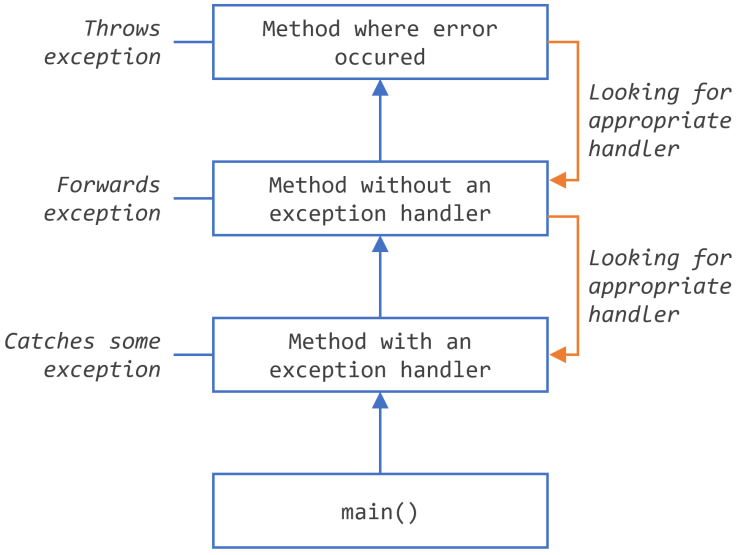
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack.



The Call Stack

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



Searching the call stack for the exception handler

Using exceptions to manage errors has some advantages over traditional error-management techniques.

4.1.2 The Catch or Specify Requirement

Valid Java programming language code must honor the *Catch or Specify Requirement*. This means that code that might throw certain exceptions must be enclosed by either of the following:

- A try statement that catches the exception. The try must provide a handler for the exception, as described in Catching and Handling Exceptions.

- A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception, as described in Specifying the Exceptions Thrown by a Method.

Code that fails to honor the Catch or Specify Requirement will not compile.

Not all exceptions are subject to the Catch or Specify Requirement. To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement.

4.1.3 The Three Kinds of Exceptions

Checked Exception

The first kind of exception is the *checked exception*. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the constructor for **java.io.FileReader**. Normally, the user provides the name of an existing, readable file, so the construction of the **FileReader** object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws **java.io.FileNotFoundException**. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

Checked exceptions are subject to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by **Error**, **RuntimeException**, and their subclasses.

Error

The second kind of exception is the error. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw **java.io.IOError**. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

Errors are not subject to the Catch or Specify Requirement. Errors are those exceptions indicated by **Error** and its subclasses.

Runtime Exception

The third kind of exception is the runtime exception. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described

previously that passes a file name to the constructor for **FileReader**. If a logic error causes a null to be passed to the constructor, the constructor will throw **NullPointerException**. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Runtime exceptions are not subject to the Catch or Specify Requirement. Runtime exceptions are those indicated by **RuntimeException** and its subclasses.

Errors and runtime exceptions are collectively known as *unchecked exceptions*.

4.1.4 Bypassing Catch or Specify

Some programmers consider the Catch or Specify Requirement a serious flaw in the exception mechanism and bypass it by using unchecked exceptions in place of checked exceptions. In general, this is not recommended. The section Unchecked Exceptions - The Controversy talks about when it is appropriate to use unchecked exceptions.

4.2 Catching and Handling Exceptions

4.2.1 Catching and Handling Exceptions

This section describes how to use the three exception handler components — the *try*, *catch*, and *finally* blocks — to write an exception handler. Then, the try-with-resources statement, introduced in Java SE 7, is explained. The try-with-resources statement is particularly suited to situations that use **Closeable** resources, such as streams.

The last part of this section walks through an example and analyzes what occurs during various scenarios.

The following example defines and implements a class named **ListOfNumbers**. When constructed, **ListOfNumbers** creates an **ArrayList** that contains 10 Integer elements with sequential values 0 through 9. The **ListOfNumbers** class also defines a method named $writeList()$, which writes the list of numbers into a text file called OutFile.txt. This example uses output classes defined in *java.io*.

```
// Note: This class will not compile yet.
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {
    private List<Integer> list;
```

```
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(i);
        }
    }

    public void writeList () {
        // The FileWriter constructor throws IOException, which must be caught.
        PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE; i++) {
            // The get(int) method throws IndexOutOfBoundsException, which must be
                ↪ caught.
            out.println("Value at: " + i + " = " + list.get(i));
        }

        out.close();
    }
}
```

The first line in boldface is a call to a constructor. The constructor initializes an output stream on a file. If the file cannot be opened, the constructor throws an **IOException**. The second boldface line is a call to the **ArrayList** class's get method, which throws an **IndexOutOfBoundsException** if the value of its argument is too small (less than 0) or too large (more than the number of elements currently contained by the **ArrayList**.

If you try to compile the **ListOfNumbers** class, the compiler prints an error message about the exception thrown by the **FileWriter** constructor. However, it does not display an error message about the exception thrown by $get()$. The reason is that the exception thrown by the constructor, **IOException**, is a checked exception, and the one thrown by the $get()$ method, **IndexOutOfBoundsException**, is an unchecked exception.

Now that you're familiar with the **ListOfNumbers** class and where the exceptions can be thrown within it, you're ready to write exception handlers to catch and handle those exceptions.

4.2.2 The Try Block

The first step in constructing an exception handler is to enclose the code that might throw an exception within a *try* block. In general, a *try* block looks like the following:

```
try {
  code
}
catch and finally blocks . . .
```

The segment in the example labeled code contains one or more legal lines of code that could throw an exception.

To construct an exception handler for the $writeList()$ method from the **ListOfNumbers** class, enclose the exception-throwing statements of the $writeList()$ method within a try block. There is more than one way to do this. You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each. Or, you can put all the $writeList()$ code within a single try block and associate multiple handlers with it. The following listing uses one try block for the entire method because the code in question is very short.

```
private List<Integer> list;
private static final int SIZE = 10;

public void writeList() {
  PrintWriter out = null;
  try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
      out.println("Value at: " + i + " = " + list.get(i));
    }
  }
  catch and finally blocks . . .
}
```

If an exception occurs within the *try* block, that exception is handled by an exception handler associated with it. To associate an exception handler with a *try* block, you must put a *catch* block after it; the next section, The *catch* blocks, shows you how.

4.2.3 The Catch Blocks

You associate exception handlers with a *try* block by providing one or more catch blocks directly after the *try* block. No code can be between the end of the *try* block and the beginning of the first catch block.

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

Each *catch* block is an exception handler that handles the type of exception indicated by its argument. The argument type, **ExceptionType**, declares the type of exception that the handler can handle and must be the name of a class that inherits from the **Throwable** class. The handler can refer to the exception with name.

The *catch* block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose **ExceptionType** matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

The following are two exception handlers for the $writeList()$ method:

```
try {

} catch (IndexOutOfBoundsException e) {
    System.err.println("IndexOutOfBoundsException: " + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: " + e.getMessage());
}
```

Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions, as described in the Chained Exceptions section.

4.2.4 Multi-Catching Exceptions

You can catch more than one type of exception with one exception handler, with the multi-catch pattern.

In Java SE 7 and later, a single *catch* block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

In the *catch* clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```

Note: If a catch block handles more than one exception type, then the *catch* parameter is implicitly *final*. In this example, the *catch* parameter ex is *final* and therefore you cannot assign any values to it within the *catch* block.

4.2.5 The Finally Block

The finally block always executes when the *try* block exits. This ensures that the *finally* block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a *return*, *continue*, or *break*. Putting cleanup code in a *finally* block is always a good practice, even when no exceptions are anticipated.

Note: If the JVM exits while the try or catch code is being executed, then the finally block may not execute.

The *try* block of the $writeList()$ method that you've been working with here opens a **PrintWriter**. The program should close that stream before exiting the $writeList()$ method. This poses a somewhat complicated problem because *try* block of $writeList()$ can exit in one of three ways.

- The new **FileWriter** statement fails and throws an **IOException**.
- The $list.get(i)$ statement fails and throws an **IndexOutOfBoundsException**.
- Everything succeeds and the *try* block exits normally.

The runtime system always executes the statements within the *finally* block regardless of what happens within the *try* block. So it's the perfect place to perform cleanup.

The following *finally* block for the $writeList()$ method cleans up and then closes the **PrintWriter**.

```
finally {
  if (out != null) {
    System.out.println("Closing PrintWriter");
    out.close();
  } else {
    System.out.println("PrintWriter not open");
  }
}
```

Important: The *finally* block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a *finally* block to ensure that resource is always recovered.

Consider using the try-with-resources statement in these situations, which automatically releases system resources when no longer needed. The try-with-resources Statement section has more information.

4.2.6 The Try-with-resources Statement

The *try-with-resources* statement is a *try* statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements **java.lang.AutoCloseable**, which includes all objects which implement **java.io.Closeable**, can be used as a resource.

The following example reads the first line from a file. It uses an instance of **BufferedReader** to read data from the file. **BufferedReader** is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {
  // Using try-with-resources statement
  try (BufferedReader br = new BufferedReader(new FileReader(path))) {
    return br.readLine();
  }
}
```

In this example, the resource declared in the try-with-resources statement is a

BufferedReader. The declaration statement appears within parentheses immediately after the try keyword. The class **BufferedReader**, in Java SE 7 and later, implements the interface **java.lang.AutoCloseable**. Because the **BufferedReader** instance is declared in a try-with-resource statement, it will be closed regardless of whether the try statement completes normally or abruptly (as a result of the method **BufferedReader**.$readLine()$ throwing an **IOException**.

Prior to Java SE 7, you can use a *finally* block to ensure that a resource is closed regardless of whether the *try* statement completes normally or abruptly.

The following example uses a *finally* block instead of a try-with-resources statement:

```
static String readFirstLineFromFileWithFinallyBlock(String path)
    throws IOException {
  BufferedReader br = new BufferedReader(new FileReader(path));
  try {
    return br.readLine();
  } finally {
    br.close();
  }
}
```

However, in this example, if the methods $readLine()$ and close both throw exceptions, then the method $readFirstLineFromFileWithFinallyBlock()$ throws the exception thrown from the *finally* block; the exception thrown from the *try* block is suppressed. In contrast, in the example $readFirstLineFromFile()$, if exceptions are thrown from both the *try* block and the try-with-resources statement, then the method $readFirstLineFromFile()$ throws the exception thrown from the *try* block; the exception thrown from the try-with-resources block is suppressed. In Java SE 7 and later, you can retrieve suppressed exceptions.

You may declare one or more resources in a try-with-resources statement. The following example retrieves the names of the files packaged in the zip file $zipFileName$ and creates a text file that contains the names of these files:

```
public static void writeToFileZipFileContents(String zipFileName,
    String outputFileName) throws java.io.IOException {
  java.nio.charset.Charset charset =
  java.nio.charset.StandardCharsets.US_ASCII;
  java.nio.file.Path outputFilePath =
  java.nio.file.Paths.get(outputFileName);

```

```
    // Open zip file and create output file with try-with-resources statement
    try (
      java.util.zip.ZipFile zf =
      new java.util.zip.ZipFile(zipFileName);
      java.io.BufferedWriter writer =
      java.nio.file.Files.newBufferedWriter(outputFilePath, charset)
    ) {
      // Enumerate each entry
      for (java.util.Enumeration entries =
          zf.entries(); entries.hasMoreElements();) {
        // Get the entry name and write it to the output file
        String newLine = System.getProperty("line.separator");
        String zipEntryName =
          ((java.util.zip.ZipEntry) entries.nextElement()).getName() +
        newLine;
        writer.write(zipEntryName, 0, zipEntryName.length());
      }
    }
  }
```

In this example, the try-with-resources statement contains two declarations that are separated by a semicolon: **ZipFile** and **BufferedWriter**. When the block of code that directly follows it terminates, either normally or because of an exception, the close() methods of the BufferedWriter and **ZipFile** objects are automatically called in this order. Note that the close methods of resources are called in the opposite order of their creation.

The following example uses a try-with-resources statement to automatically close a **java.sql.Statement** object:

```
public static void viewTable(Connection con) throws SQLException {
  String query = "SELECT cof_name, sup_id, price, sales, total FROM coffees";

  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
      String coffeeName = rs.getString("cof_name");
      int supplierId = rs.getInt("sup_id");
      float price = rs.getFloat("price");
      int sales = rs.getInt("sales");
      int total = rs.getInt("total");

      System.out.println(coffeeName + ", " + supplierId + ", " +
      price + ", " + sales + ", " + total);
    }
```

```
    } catch (SQLException e) {
        JDBCTutorialUtilities .printSQLException(e);
    }
}
```

The resource **java.sql.Statement** used in this example is part of the JDBC 4.1 and later API.

Note: A try-with-resources statement can have *catch* and *finally* blocks just like an ordinary try statement. In a try-with-resources statement, any *catch* or *finally* block is run after the resources declared have been closed.

4.2.7 Suppressed Exceptions

An exception can be thrown from the block of code associated with the try-with-resources statement. In the example $writeToFileZipFileContents()$, an exception can be thrown from the try block, and up to two exceptions can be thrown from the try-with-resources statement when it tries to close the **ZipFile** and **BufferedWriter** objects. If an exception is thrown from the *try* block and one or more exceptions are thrown from the try-with-resources statement, then those exceptions thrown from the try-with-resources statement are suppressed, and the exception thrown by the block is the one that is thrown by the $writeToFileZipFileContents()$ method. You can retrieve these suppressed exceptions by calling the **Throwable**.$getSuppressed()$ method from the exception thrown by the *try* block.

4.2.8 Classes That Implement the AutoCloseable or Closeable Interface

See the Javadoc of the **AutoCloseable** and **Closeable** interfaces for a list of classes that implement either of these interfaces. The **Closeable** interface extends the **AutoCloseable** interface. The $close()$ method of the **Closeable** interface throws exceptions of type **IOException** while the $close()$ method of the **AutoCloseable** interface throws exceptions of type **Exception**. Consequently, subclasses of the **AutoCloseable** interface can override this behavior of the $close()$ method to throw specialized exceptions, such as **IOException**, or no exception at all.

4.2.9 Putting It All Together

The previous sections described how to construct the *try*, *catch*, and *finally* code blocks for the $writeList()$ method in the **ListOfNumbers** class. Now, let's walk through the code and investigate what can happen.

When all the components are put together, the $writeList()$ method looks like the following.

```
1  public void writeList() {
     PrintWriter out = null;
3    try {
       System.out.println("Entering" + " try statement");
5      out = new PrintWriter(new FileWriter("OutFile.txt"));
       for (int i = 0; i < SIZE; i++) {
7        out.println("Value at: " + i + " = " + list.get(i));
       }
9    } catch (IndexOutOfBoundsException e) {
       System.err.println("Caught IndexOutOfBoundsException: " + e.getMessage());
11   } catch (IOException e) {
       System.err.println("Caught IOException: " + e.getMessage());
13   } finally {
       if (out != null) {
15       System.out.println("Closing PrintWriter");
         out.close();
17     } else {
         System.out.println("PrintWriter not open");
19     }
     }
21 }
```

As mentioned previously, this method's *try* block has three different exit possibilities; here are two of them.

- Code in the try statement fails and throws an exception. This could be an **IOException** caused by the new **FileWriter** statement or an **IndexOutOfBoundsException** caused by a wrong index value in the for loop.
- Everything succeeds and the try statement exits normally.

Let's look at what happens in the $writeList()$ method during these two exit possibilities.

Scenario 1: An Exception Occurs

The statement that creates a **FileWriter** can fail for a number of reasons. For example, the constructor for the **FileWriter** throws an **IOException** if the program cannot create or write to the file indicated.

When **FileWriter** throws an **IOException**, the runtime system immediately stops executing the *try* block; method calls being executed are not completed. The runtime system then starts searching at the top of the method call stack for an appropriate exception handler. In this example, when the **IOException** occurs, the

FileWriter constructor is at the top of the call stack. However, the **FileWriter** constructor doesn't have an appropriate exception handler, so the runtime system checks the next method — the $writeList()$ method — in the method call stack. The $writeList()$ method has two exception handlers: one for **IOException** and one for **IndexOutOfBoundsException**.

The runtime system checks $writeList()$'s handlers in the order in which they appear after the try statement. The argument to the first exception handler is **IndexOutOfBoundsException**. This does not match the type of exception thrown, so the runtime system checks the next exception handler — **IOException**. This matches the type of exception that was thrown, so the runtime system ends its search for an appropriate exception handler. Now that the runtime has found an appropriate handler, the code in that catch block is executed.

After the exception handler executes, the runtime system passes control to the finally block. Code in the finally block executes regardless of the exception caught above it. In this scenario, the **FileWriter** was never opened and doesn't need to be closed. After the *finally* block finishes executing, the program continues with the first statement after the *finally* block.

Here's the complete output from the **ListOfNumbers** program that appears when an **IOException** is thrown.

Command window

```
Entering try statement
Caught IOException: OutFile.txt
PrintWriter not open
```

Scenario 2: The try Block Exits Normally

In this scenario, all the statements within the scope of the *try* block execute successfully and throw no exceptions. Execution falls off the end of the *try* block, and the runtime system passes control to the *finally* block. Because everything was successful, the **PrintWriter** is open when control reaches the *finally* block, which closes the **PrintWriter**. Again, after the *finally* block finishes executing, the program continues with the first statement after the *finally* block.

Here is the output from the **ListOfNumbers** program when no exceptions are thrown.

Command window

```
Entering try statement
Closing PrintWriter
```

4.3 Throwing Exceptions

4.3.1 Specifying the Exceptions Thrown by a Method

The previous section showed how to write an exception handler for the *writeList()* method in the **ListOfNumbers** class. Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example, if you were providing the **ListOfNumbers** class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package. In this case, it's better to not catch the exception and to allow a method further up the call stack to handle it.

If the *writeList()* method doesn't catch the checked exceptions that can occur within it, the *writeList()* method must specify that it can throw these exceptions. Let's modify the original *writeList()* method to specify the exceptions it can throw instead of catching them. To remind you, here's the original version of the *writeList()* method that won't compile.

```
public void writeList() {
  PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));
  for (int i = 0; i < SIZE; i++) {
    out.println("Value at: " + i + " = " + list.get(i));
  }
  out.close();
}
```

To specify that *writeList()* can throw two exceptions, add a throws clause to the method declaration for the *writeList()* method. The *throws* clause comprises the *throws* keyword followed by a comma-separated list of all the exceptions thrown by that method. The clause goes after the method name and argument list and before the brace that defines the scope of the method; here's an example.

```
public void writeList() throws IOException, IndexOutOfBoundsException {
    .....
}
```

Remember that **IndexOutOfBoundsException** is an unchecked exception; including it in the *throws* clause is not mandatory. You could just write the following.

```
public void writeList () throws IOException {
    .....
}
```

4.3.2 How to Throw Exceptions

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the *throw* statement.

As you have probably noticed, the Java platform provides numerous exception classes. All the classes are descendants of the **Throwable** class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

You can also create your own exception classes to represent problems that can occur within the classes you write. In fact, if you are a package developer, you might have to create your own set of exception classes to allow users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages.

You can also create chained exceptions. For more information, see the Chained Exceptions section.

4.3.3 The Throw Statement

All methods use the *throw* statement to throw an exception. The *throw* statement requires a single argument: a throwable object. **Throwable** objects are instances of any subclass of the Throwable class. Here's an example of a *throw* statement.

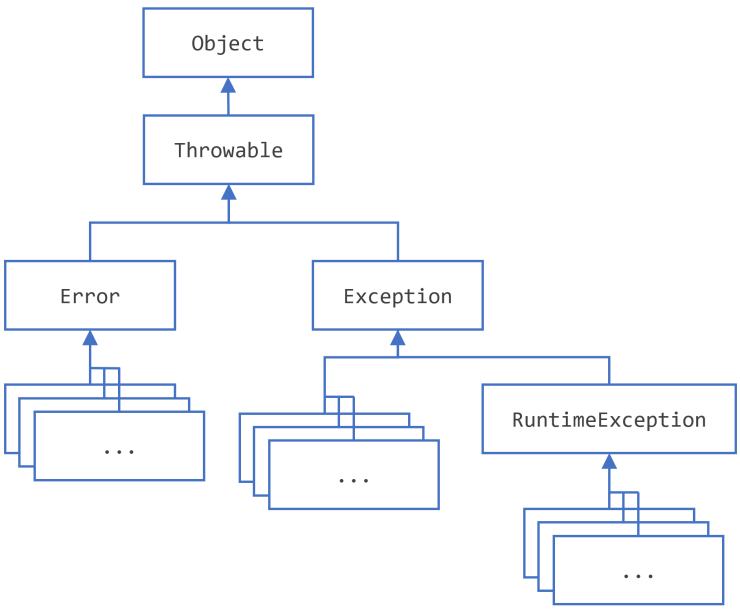
```
throw someThrowableObject;
```

Let's look at the *throw* statement in context. The following $pop()$ method is taken from a class that implements a common stack object. The method removes the top element from the stack and returns the object. Note that the declaration of the $pop()$ method does not contain a throws clause. **EmptyStackException** is not a checked exception, so pop is not required to state that it might occur.

```
public Object pop() {
  if (size == 0) {
    throw new EmptyStackException();
  }

  Object obj = objectAt(size - 1);
  setObjectAt(size - 1, null);
  size--;

  return obj;
}
```

The *pop*() method checks to see whether any elements are on the stack. If the stack is empty (its size is equal to 0), pop instantiates a new **EmptyStackException** object, a member of *java.util* and throws it. The Creating Exception Classes section explains how to create your own exception classes. For now, all you need to remember is that you can throw only objects that inherit from the **java.lang.Throwable** class.

4.3.4 Throwable Class and Its Subclasses



The Throwable hierarchy

The objects that inherit from the **Throwable** class include direct descendants (objects that inherit directly from the **Throwable** class) and indirect descendants

(objects that inherit from children or grandchildren of the **Throwable** class). The figure below illustrates the class hierarchy of the Throwable class and its most significant subclasses. As you can see, **Throwable** has two direct descendants: **Error** and **Exception**.

4.3.5 Error Class

When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an **Error**. Simple programs typically do not catch or throw instances of **Error**.

4.3.6 Exception Class

Most programs throw and catch objects that derive from the **Exception** class. An **Exception** indicates that a problem occurred, but it is not a serious system problem. Most programs you write will throw and catch instances of Exception as opposed to **Error**.

The Java platform defines the many descendants of the **Exception** class. These descendants indicate various types of exceptions that can occur. For example, **IllegalAccessException** signals that a particular method could not be found, and **NegativeArraySizeException** indicates that a program attempted to create an array with a negative size.

One **Exception** subclass, **RuntimeException**, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception is **NullPointerException**, which occurs when a method tries to access a member of an object through a null reference. The section Unchecked Exceptions — The Controversy discusses why most applications shouldn't throw runtime exceptions or subclass **RuntimeException**.

4.3.7 Chained Exceptions

An application often responds to an exception by throwing another exception. In effect, the first exception causes the second exception. It can be very helpful to know when one exception causes another. Chained Exceptions help the programmer do this.

The following are the methods and constructors in Throwable that support chained exceptions.

```
Throwable getCause()
Throwable initCause(Throwable)
```

```
Throwable(String, Throwable)
Throwable(Throwable)
```

The **Throwable** argument to $initCause()$ and the **Throwable** constructors is the exception that caused the current exception. $getCause()$ returns the exception that caused the current exception, and $initCause()$ sets the current exception's cause.

The following example shows how to use a chained exception.

```
try {

} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

In this example, when an **IOException** is caught, a new **SampleException** exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher level exception handler.

4.3.8 Accessing Stack Trace Information

Now let's suppose that the higher-level exception handler wants to dump the stack trace in its own format.

The following code shows how to call the $getStackTrace()$ method on the exception object.

```
catch (Exception cause) {
    StackTraceElement elements[] = cause.getStackTrace();
    for (int i = 0, n = elements.length; i < n; i++) {
        System.err.println(elements[i].getFileName()
            + ":" + elements[i].getLineNumber()
            + ">> "
            + elements[i].getMethodName() + "()");
    }
}
```

Logging API

The next code snippet logs where an exception occurred from within the catch block. However, rather than manually parsing the stack trace and sending the output to *java.util.logging*, it sends the output to a file using the logging facility in the *java.util.logging* package.

```
try {
  Handler handler = new FileHandler("OutFile.log");
  Logger.getLogger("").addHandler(handler);
} catch (IOException e) {
  Logger logger = Logger.getLogger("package.name");
  StackTraceElement elements[] = e.getStackTrace();
  for (int i = 0, n = elements.length; i < n; i++) {
    logger.log(Level.WARNING, elements[i].getMethodName());
  }
}
```

4.3.9 Creating Exception Classes

When faced with choosing the type of exception to throw, you can either use one written by someone else — the Java platform provides a lot of exception classes you can use — or you can write one of your own. You should write your own exception classes if you answer yes to any of the following questions; otherwise, you can probably use someone else's.

- Do you need an exception type that isn't represented by those in the Java platform?
- Would it help users if they could differentiate your exceptions from those thrown by classes written by other vendors?
- Does your code throw more than one related exception?
- If you use someone else's exceptions, will users have access to those exceptions? A similar question is, should your package be independent and self-contained?

An Example

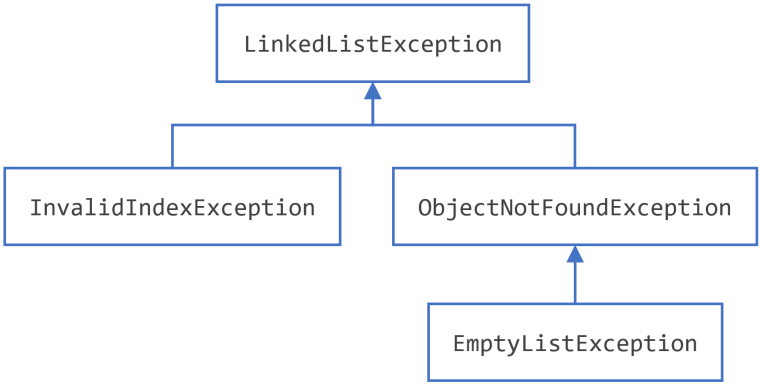
Suppose you are writing a linked list class. The class supports the following methods, among others:

- *objectAt*(*int n*) — Returns the object in the nth position in the list. Throws an exception if the argument is less than 0 or more than the number of objects currently in the list.

- $firstObject()$ — Returns the first object in the list. Throws an exception if the list contains no objects.
- $indexOf(Object\ o)$ — Searches the list for the specified object and returns its position in the list. Throws an exception if the object passed into the method is not in the list.

The linked list class can throw multiple exceptions, and it would be convenient to be able to catch all exceptions thrown by the linked list with one exception handler. Also, if you plan to distribute your linked list in a package, all related code should be packaged together. Thus, the linked list should provide its own set of exception classes.

The next figure illustrates one possible class hierarchy for the exceptions thrown by the linked list.



Example of exception class hierarchy

Choosing a Superclass

Any **Exception** subclass can be used as the parent class of **LinkedListException**. However, a quick perusal of those subclasses shows that they are inappropriate because they are either too specialized or completely unrelated to **LinkedListException**. Therefore, the parent class of **LinkedListException** should be **Exception**.

Most applications you write will throw objects that are instances of **Exception**. Instances of **Error** are normally used for serious, hard errors in the system, such as those that prevent the JVM from running.

Note: For readable code, it's good practice to append the string **Exception** to the names of all classes that inherit (directly or indirectly) from the **Exception** class.

4.4 Unchecked Exceptions — The Controversy

4.4.1 Unchecked Exceptions — The Controversy

Because the Java programming language does not require methods to catch or to specify unchecked exceptions (**RuntimeException**, **Error**, and their subclasses), programmers may be tempted to write code that throws only unchecked exceptions or to make all their exception subclasses inherit from **RuntimeException**. Both of these shortcuts allow programmers to write code without bothering with compiler errors and without bothering to specify or to catch any exceptions. Although this may seem convenient to the programmer, it sidesteps the intent of the catch or specify requirement and can cause problems for others using your classes.

Why did the designers decide to force a method to specify all uncaught checked exceptions that can be thrown within its scope? Any **Exception** that can be thrown by a method is part of the method's public programming interface. Those who call a method must know about the exceptions that a method can throw so that they can decide what to do about them. These exceptions are as much a part of that method's programming interface as its parameters and return value.

The next question might be: "If it's so good to document a method's API, including the exceptions it can throw, why not specify runtime exceptions too?" Runtime exceptions represent problems that are the result of a programming problem, and as such, the API client code cannot reasonably be expected to recover from them or to handle them in any way. Such problems include arithmetic exceptions, such as dividing by zero; pointer exceptions, such as trying to access an object through a null reference; and indexing exceptions, such as attempting to access an array element through an index that is too large or too small.

Runtime exceptions can occur anywhere in a program, and in a typical one they can be very numerous. Having to add runtime exceptions in every method declaration would reduce a program's clarity. Thus, the compiler does not require that you catch or specify runtime exceptions (although you can).

One case where it is common practice to throw a **RuntimeException** is when the user calls a method incorrectly. For example, a method can check if one of its arguments is incorrectly null. If an argument is null, the method might throw a **NullPointerException**, which is an unchecked exception.

Generally speaking, do not throw a **RuntimeException** or create a subclass of **RuntimeException** simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Here's the bottom line guideline: If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

4.4.2 Advantages of Exceptions

Now that you know what exceptions are and how to use them, it's time to learn the advantages of using exceptions in your programs.

Advantage 1: Separating Error-Handling Code from "Regular" Code

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code. For example, consider the pseudocode method here that reads an entire file into memory.


```
// Process of reading an entire file into memory
readFile {
  open the file;
  determine its size;
  allocate that much memory;
  read the file into memory;
  close the file;
}
```

At first glance, this function seems simple enough, but it ignores all the following potential errors.

- What happens if the file can't be opened?
- What happens if the length of the file can't be determined?
- What happens if enough memory can't be allocated?
- What happens if the read fails?
- What happens if the file can't be closed?

To handle such cases, the $readFile()$ function must have more code to do error detection, reporting, and handling. Here is an example of what the function might look like.

PSEUDO CODE

```
// There's so much error detection, reporting, and returning
errorCodeType readFile {
  initialize errorCode = 0;

  open the file;
  if (theFileIsOpen) {
    determine the length of the file;
    if (gotTheFileLength) {
      allocate that much memory;
      if (gotEnoughMemory) {
        read the file into memory;
        if (readFailed) {
          errorCode = -1;
        }
      } else {
        errorCode = -2;
      }
    } else {
      errorCode = -3;
    }

    close the file;
    if (theFileDidntClose && errorCode == 0) {
      errorCode = -4;
    } else {
      errorCode = errorCode and -4;
    }
  } else {
    errorCode = -5;
  }

  return errorCode;
}
```

There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. Worse yet, the logical flow of the code has also been lost, thus making it difficult to tell whether the code is doing the right thing: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing when you modify the method three months after writing it. Many programmers solve this problem by simply ignoring it — errors are reported when their programs crash.

Exceptions enable you to write the main flow of your code and to deal with the exceptional cases elsewhere. If the $readFile()$ function used exceptions instead of traditional error-management techniques, it would look more like the following.

```
// Exceptions do help you organize the work more effectively .
readFile {
  try {
    open the file ;
    determine its size ;
     allocate that much memory;
    read the file into memory;
     close the file ;
  } catch ( fileOpenFailed ) {
    doSomething;
  } catch ( sizeDeterminationFailed ) {
    doSomething;
  } catch (memoryAllocationFailed) {
    doSomething;
  } catch ( readFailed ) {
    doSomething;
  } catch ( fileCloseFailed ) {
    doSomething;
  }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

Advantage 2: Propagating Errors Up the Call Stack

A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the $readFile()$ method is the fourth method in a series of nested method calls made by the main program: $method1()$ calls $method2()$, which calls $method3()$, which finally calls $readFile()$.

```
method1 {
    call method2;
}

method2 {
    call method3;
}

method3 {
    call readFile ;
}
```

Suppose also that $method1$ is the only method interested in the errors that might occur within $readFile()$. Traditional error-notification techniques force $method2()$ and $method3()$ to propagate the error codes returned by $readFile()$ up the call stack until the error codes finally reach $method1()$—the only method that is interested in them.

```
method1 {
  errorCodeType error;
  error = call method2;
  if (error)
    doErrorProcessing;
  else
    proceed;
}

errorCodeType method2 {
  errorCodeType error;
  error = call method3;
  if (error)
    return error;
  else
    proceed;
}

errorCodeType method3 {
  errorCodeType error;
  error = call readFile;
  if (error)
    return error;
  else
    proceed;
}
```

Recall that the Java runtime environment searches backward through the call stack to find any methods that are interested in handling a particular exception. A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors.

```
// Only the methods that care about errors have to worry about detecting errors.
method1 {
  try {
    call method2;
```

```
    } catch (exception e) {
6       doErrorProcessing;
    }
8 }

10 method2 throws exception {
      call method3;
12 }

14 method3 throws exception {
      call readFile ;
16 }
```

However, as the pseudocode shows, ducking an exception requires some effort on the part of the middleman methods. Any checked exceptions that can be thrown within a method must be specified in its *throws* clause.

Advantage 3: Grouping and Differentiating Error Types

Because all exceptions thrown within a program are objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy. An example of a group of related exception classes in the Java platform are those defined in *java.io* — **IOException** and its descendants. **IOException** is the most general and represents any type of error that can occur when performing I/O. Its descendants represent more specific errors. For example, **FileNotFoundException** means that a file could not be located on disk.

A method can write specific handlers that can handle a very specific exception. The **FileNotFoundException** class has no descendants, so the following handler can handle only one type of exception.

```
catch (FileNotFoundException e) {
2     .....
}
```

A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement. For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an **IOException** argument.

```
catch (IOException e) {
    .....
}
```

This handler will be able to catch all I/O exceptions, including **FileNotFoundException**, **EOFException**, and so on. You can find details about what occurred by querying the argument passed to the exception handler. For example, use the following to print the stack trace.

```
catch (IOException e) {
    // Output goes to System.err.
    e.printStackTrace();
    // Send trace to stdout.
    e.printStackTrace(System.out);
}
```

You could even set up an exception handler that handles any **Exception** with the handler here.

```
// A (too) general exception handler
catch (Exception e) {
    .....
}
```

The **Exception** class is close to the top of the **Throwable** class hierarchy. Therefore, this handler will catch many other exceptions in addition to those that the handler is intended to catch. You may want to handle exceptions this way if all you want your program to do, for example, is print out an error message for the user and then exit.

In most situations, however, you want exception handlers to be as specific as possible. The reason is that the first thing a handler must do is determine what type of exception occurred before it can decide on the best recovery strategy. In effect, by not catching specific errors, the handler must accommodate any possibility. Exception handlers that are too general can make code more error-prone by catching and handling exceptions that weren't anticipated by the programmer and for which the handler was not intended.

As noted, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

4.4.3 Summary

A program can use exceptions to indicate that an error occurred. To throw an exception, use the throw statement and provide it with an exception object — a descendant of **Throwable** — to provide information about the specific error that occurred. A method that throws an uncaught, checked exception must include a throws clause in its declaration.

A program can catch exceptions by using a combination of the *try*, *catch*, and *finally* blocks.

- The *try* block identifies a block of code in which an exception can occur.
- The *catch* block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The *finally* block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the try block.

The *try* statement should contain at least one catch block or a *finally* block and may have multiple *catch* blocks.

The class of the exception object indicates the type of exception thrown. The exception object can contain further information about the error, including an error message. With exception chaining, an exception can point to the exception that caused it, which can in turn point to the exception that caused it, and so on.

5 Analysis of Algorithms

5.1 Analysis of Algorithms

The term "efficiency" can refer to efficient use of almost any resource, including time, computer memory, disk space, or network bandwidth. However, we will deal exclusively with time efficiency, and the major question that we want to ask about a program is, how long does it take to perform its task?

It really makes little sense to classify an individual program as being "efficient" or "inefficient." It makes more sense to compare two (correct) programs that perform the same task and ask which one of the two is "more efficient," that is, which one performs the task more quickly. However, even here there are difficulties. The running time of a program is not well-defined. The run time can be different depending on the number and speed of the processors in the computer on which it is run and, in the case of Java, on the design of the Java Virtual Machine which is used to interpret the program. It can depend on details of the compiler which is used to translate the program from high-level language to machine language. Furthermore, the run time of a program depends on the size of the problem which the program has to solve. It takes a sorting program longer to sort 10000 items than it takes it to sort 100 items. When the run times of two programs are compared, it often happens that Program A solves small problems faster than Program B, while Program B solves large problems faster than Program A, so that it is simply not the case that one program is faster than the other in all cases.

In spite of these difficulties, there is a field of computer science dedicated to analyzing the efficiency of programs. The field is known as Analysis of Algorithms. The focus is on algorithms, rather than on programs as such, to avoid having to deal with multiple implementations of the same algorithm written in different languages, compiled with different compilers, and running on different computers. Analysis of Algorithms is a mathematical field that abstracts away from these down-and-dirty details. Still, even though it is a theoretical field, every working programmer should be aware of some of its techniques and results.

One of the main techniques of analysis of algorithms is asymptotic analysis. The term "asymptotic" here means basically "the tendency in the long run, as the size of the input is increased." An asymptotic analysis of an algorithm's run time looks at the question of how the run time depends on the size of the problem. The analysis is asymptotic because it only considers what happens to the run time as the size of the problem increases without limit; it is not concerned with what happens for problems

of small size or, in fact, for problems of any fixed finite size. Only what happens in the long run, as the problem size increases without limit, is important. Showing that Algorithm A is asymptotically faster than Algorithm B doesn't necessarily mean that Algorithm A will run faster than Algorithm B for problems of size 10 or size 1000 or even size 1000000—it only means that if you keep increasing the problem size, you will eventually come to a point where Algorithm A is faster than Algorithm B. An asymptotic analysis is only a first approximation, but in practice it often gives important and useful information.

5.1.1 Popular Notations in Complexity Analysis of Algorithms

Central to asymptotic analysis is Big-Oh notation. Using this notation, we might say, for example, that an algorithm has a running time that is $O(n^2)$ or $O(n)$ or $O(\log(n))$. These notations are read "Big-Oh of n squared," "Big-Oh of n," and "Big-Oh of log n" (where log is a logarithm function). More generally, we can refer to $O(f(n))$ ("Big-Oh of f of n"), where $f(n)$ is some function that assigns a positive real number to every positive integer n. The "n" in this notation refers to the size of the problem. Before you can even begin an asymptotic analysis, you need some way to measure problem size. Usually, this is not a big issue. For example, if the problem is to sort a list of items, then the problem size can be taken to be the number of items in the list. When the input to an algorithm is an integer, as in the case of an algorithm that checks whether a given positive integer is prime, the usual measure of the size of a problem is the number of bits in the input integer rather than the integer itself. More generally, the number of bits in the input to a problem is often a good measure of the size of the problem.

To say that the running time of an algorithm is $O(f(n))$ means that for large values of the problem size, n, the running time of the algorithm is no bigger than some constant times $f(n)$. (More rigorously, there is a number C and a positive integer M such that whenever n is greater than M, the run time is less than or equal to $C * f(n)$.) The constant takes into account details such as the speed of the computer on which the algorithm is run; if you use a slower computer, you might have to use a bigger constant in the formula, but changing the constant won't change the basic fact that the run time is $O(f(n))$. The constant also makes it unnecessary to say whether we are measuring time in seconds, years, CPU cycles, or any other unit of measure; a change from one unit of measure to another is just multiplication by a constant. Note also that $O(f(n))$ doesn't depend at all on what happens for small problem sizes, only on what happens in the long run as the problem size increases without limit.

To look at a simple example, consider the problem of adding up all the numbers in an array. The problem size, n, is the length of the array. Using arr as the name of the array, the algorithm can be expressed in Java as:

```
total = 0;
for (int i = 0; i < n; i++) {
   total = total + arr[i];
}
```

This algorithm performs the same operation, $total = total + arr[i]$, n times. The total time spent on this operation is $a * n$, where a is the time it takes to perform the operation once. Now, this is not the only thing that is done in the algorithm. The value of i is incremented and is compared to n each time through the loop. This adds an additional time of $b * n$ to the run time, for some constant b. Furthermore, i and total both have to be initialized to zero; this adds some constant amount c to the running time. The exact running time would then be $(a + b) * n + c$, where the constants a, b, and c depend on factors such as how the code is compiled and what computer it is run on. Using the fact that c is less than or equal to $c * n$ for any positive integer n, we can say that the run time is less than or equal to $(a+b+c) * n$. That is, the run time is less than or equal to a constant times n. By definition, this means that the run time for this algorithm is $O(n)$.

If this explanation is too mathematical for you, we can just note that for large values of n, the c in the formula $(a + b) * n + c$ is insignificant compared to the other term, $(a + b) * n$. We say that c is a "lower order term." When doing asymptotic analysis, lower order terms can be discarded. A rough, but correct, asymptotic analysis of the algorithm would go something like this: Each iteration of the for loop takes a certain constant amount of time. There are n iterations of the loop, so the total run time is a constant times n, plus lower order terms (to account for the initialization). Disregarding lower order terms, we see that the run time is $O(n)$.

Note that to say that an algorithm has run time $O(f(n))$ is to say that its run time is no bigger than some constant times $f(n)$ (for large values of n). $O(f(n))$ puts an upper limit on the run time. However, the run time could be smaller, even much smaller. For example, if the run time is $O(n)$, it would also be correct to say that the run time is $O(n^2)$ or even $O(n^{10})$. If the run time is less than a constant times n, then it is certainly less than the same constant times n^2 or n^{10}.

Of course, sometimes it's useful to have a lower limit on the run time. That is, we want to be able to say that the run time is greater than or equal to some constant times $f(n)$ (for large values of n). The notation for this is $\Omega(f(n))$, read "Omega of f of n" or "Big Omega of f of n." "Omega" is the name of a letter in the Greek alphabet, and Ω is the upper case version of that letter. (To be technical, saying that the run time of an algorithm is $\Omega(f(n))$ means that there is a positive number C and a positive integer M such that whenever n is greater than M, the run time is greater than or equal to $C * f(n)$.) $O(f(n))$ tells you something about the maximum

amount of time that you might have to wait for an algorithm to finish; $\Omega(f(n))$ tells you something about the minimum time.

The algorithm for adding up the numbers in an array has a run time that is $\Omega(n)$ as well as $O(n)$. When an algorithm has a run time that is both $\Omega(f(n))$ and $O(f(n))$, its run time is said to be $\Theta(f(n))$, read "Theta of f of n" or "Big Theta of f of n." (Theta is another letter from the Greek alphabet.) To say that the run time of an algorithm is $\Theta(f(n))$ means that for large values of n, the run time is between $a * f(n)$ and $b * f(n)$, where a and b are constants (with b greater than a, and both greater than 0).

Let's look at another example. Consider the algorithm that can be expressed in Java in the following method:

```java
/**
 * Sorts the n array elements arr[0], arr[1], ..., arr[n-1] into increasing order.
 */
public static void simpleBubbleSort(int[] arr) {
  for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr.length - 1; j++) {
      if (arr[j] > arr[j + 1]) {
        swap(arr, j, j + 1)
      }
    }
  }
}

public static void swap(int[] arr, int i, int j) {
  int temp = arr[i];
  arr[i] = arr[j];
  arr[j] = temp;
}
```

Here, the parameter n represents the problem size. The outer for loop in the method is executed n times. Each time the outer for loop is executed, the inner for loop is executed $n - 1$ times, so the if statement is executed $n * (n - 1)$ times. This is $n^2 - n$, but since lower order terms are not significant in an asymptotic analysis, it's good enough to say that the if statement is executed about n^2 times. In particular, the test $arr[j] > arr[j + 1]$ is executed about n^2 times, and this fact by itself is enough to say that the run time of the algorithm is $\Omega(n^2)$, that is, the run time is at least some constant times n^2. Furthermore, if we look at other operations—the assignment statements, incrementing i and j, etc. none of them are executed more than n^2 times, so the run time is also $O(n^2)$, that is, the run time is no more than

some constant times n^2. Since it is both $\Omega(n^2)$ and $O(n^2)$, the run time of the simpleBubbleSort algorithm is $\Theta(n^2)$.

You should be aware that some people use the notation $O(f(n))$ as if it meant $\Theta(f(n))$. That is, when they say that the run time of an algorithm is $O(f(n))$, they mean to say that the run time is about equal to a constant times $f(n)$. For that, they should use $\Theta(f(n))$. Properly speaking, $O(f(n))$ means that the run time is less than a constant times $f(n)$, possibly much less.

5.1.2 Measurement of Complexity of an Algorithm

So far, my analysis has ignored an important detail. We have looked at how run time depends on the problem size, but in fact the run time usually depends not just on the size of the problem but on the specific data that has to be processed. For example, the run time of a sorting algorithm can depend on the initial order of the items that are to be sorted, and not just on the number of items.

To account for this dependency, we can consider either the worst case run time analysis or the average case run time analysis of an algorithm. For a worst case run time analysis, we consider all possible problems of size n and look at the longest possible run time for all such problems. For an average case analysis, we consider all possible problems of size n and look at the average of the run times for all such problems. Usually, the average case analysis assumes that all problems of size n are equally likely to be encountered, although this is not always realistic - or even possible in the case where there is an infinite number of different problems of a given size.

In many cases, the average and the worst case run times are the same to within a constant multiple. This means that as far as asymptotic analysis is concerned, they are the same. That is, if the average case run time is $O(f(n))$ or $\Theta(f(n))$, then so is the worst case. However, the average and worst case asymptotic analyses can differ.

It is also possible to talk about best case run time analysis, which looks at the shortest possible run time for all inputs of a given size. However, a best case analysis is only occasionally useful.

5.1.3 Rate of Growth

So, what do you really have to know about analysis of algorithms to read the rest of this book? We will not do any rigorous mathematical analysis, but you should be able to follow informal discussion of simple cases such as the examples that we have looked at in this section. Most important, though, you should have a feeling for exactly what it means to say that the running time of an algorithm is $O(f(n))$ or $\Theta(f(n))$ for some common functions $f(n)$. The main point is that these notations do not tell you anything about the actual numerical value of the running time of

the algorithm for any particular case. They do not tell you anything at all about the running time for small values of n. What they do tell you is something about the rate of growth of the running time as the size of the problem increases.

Suppose you compare two algorithms that solve the same problem. The run time of one algorithm is $\Theta(n^2)$, while the run time of the second algorithm is $\Theta(n^3)$. What does this tell you? If you want to know which algorithm will be faster for some particular problem of size, say, 100, nothing is certain. As far as you can tell just from the asymptotic analysis, either algorithm could be faster for that particular case—or in any particular case. But what you can say for sure is that if you look at larger and larger problems, you will come to a point where the $\Theta(n^2)$ algorithm is faster than the $\Theta(n^3)$ algorithm. Furthermore, as you continue to increase the problem size, the relative advantage of the $\Theta(n^2)$ algorithm will continue to grow. There will be values of n for which the $\Theta(n^2)$ algorithm is a thousand times faster, a million times faster, a billion times faster, and so on. This is because for any positive constants a and b, the function $a * n^3$ grows faster than the function $b * n^2$ as n gets larger. (Mathematically, the limit of the ratio of $a * n^3$ to $b * n^2$ is infinite as n approaches infinity.)

This means that for "large" problems, a $\Theta(n^2)$ algorithm will definitely be faster than a $\Theta(n^3)$ algorithm. You just don't know - based on the asymptotic analysis alone - exactly how large "large" has to be. In practice, in fact, it is likely that the $\Theta(n^2)$ algorithm will be faster even for fairly small values of n, and absent other information you would generally prefer a $\Theta(n^2)$ algorithm to a $\Theta(n^3)$ algorithm.

So, to understand and apply asymptotic analysis, it is essential to have some idea of the rates of growth of some common functions. For the power functions n, n^2, n^3, n^4, . . . , the larger the exponent, the greater the rate of growth of the function. Exponential functions such as $2n$ and $10n$, where the n is in the exponent, have a growth rate that is faster than that of any power function. In fact, exponential functions grow so quickly that an algorithm whose run time grows exponentially is almost certainly impractical even for relatively modest values of n, because the running time is just too long. Another function that often turns up in asymptotic analysis is the logarithm function, $\log(n)$. There are actually many different logarithm functions, but the one that is usually used in computer science is the so-called logarithm to the base two, which is defined by the fact that $\log(2^x) = x$ for any number x. (Usually, this function is written $\log_2(n)$, but I will leave out the subscript 2, since I will only use the base-two logarithm in this documentation.) The logarithm function grows very slowly. The growth rate of $\log(n)$ is much smaller than the growth rate of n. The growth rate of $n * \log(n)$ is a little larger than the growth rate of n, but much smaller than the growth rate of n^2. The following table should help you understand the differences among the rates of growth of various functions:

n	log(n)	n*log(n)	n^2	n / log(n)
16	4	64	256	4.0
64	6	384	4096	10.7
256	8	2048	65536	32.0
1024	10	10240	1048576	102.4
1000000	20	19931568	1000000000000	50173.1
1000000000	30	29897352854	1000000000000000000	33447777.3

The reason that $\log(n)$ shows up so often is because of its association with multiplying and dividing by two: Suppose you start with the number n and divide it by 2, then divide by 2 again, and so on, until you get a number that is less than or equal to 1. Then the number of divisions is equal (to the nearest integer) to $\log(n)$.

As an example, consider the binary search algorithm. This algorithm searches for an item in a sorted array. The problem size, n, can be taken to be the length of the array. Each step in the binary search algorithm divides the number of items still under consideration by 2, and the algorithm stops when the number of items under consideration is less than or equal to 1 (or sooner). It follows that the number of steps for an array of length n is at most $\log(n)$. This means that the worst-case run time for binary search is $\Theta(log(n))$. (The average case run time is also $\Theta(log(n))$.) By comparison, the linear search algorithm, which has a run time that is $\Theta(n)$. The Θ notation gives us a quantitative way to express and to understand the fact that binary search is "much faster" than linear search.

In binary search, each step of the algorithm divides the problem size by 2. It often happens that some operation in an algorithm (not necessarily a single step) divides the problem size by 2. Whenever that happens, the logarithm function is likely to show up in an asymptotic analysis of the run time of the algorithm.

Analysis of Algorithms is a large, fascinating field. It can be very helpful for understanding the differences among algorithms.

Part V Design Patterns

Part VI Java Generics

Part VII Java Concurrency

Part VIII Java GUI

Part IX References