By Gabriel Vainer and Juefei Lu

Single Responsibility Principle: *A class should only have one reason to change*
Design Smell: *rigidity , multiple responsibilities (duh)*

To clarify, a responsibility **is not** a method, meaning that SRP doesn't just mean "one method per class". A responsibility in regards to SOLID is simply just a group of methods that accomplish a particular behavior. However, SRP implies that there is an effective way of achieving this behaviour. Furthermore, the main idea SRP talks about is the idea of ***cohesion*** when designing our software.

But what exactly is cohesion? Really, it's just a matter of making sure that things that are related are grouped together, and things that aren't related are not. With that being said the Single Responsibility Principle states that we should group only those things that satisfy a single responsibility. Lets go over a quick example:

```
class Transaction {
   private void Buy(String stock, int quantity, float price){
   }
   private void Sell(String stock, int quantity, float price){
   }
}
```

Let's say we have a simple class that takes care of making transactions. Not so bad right? Everything seems to be in order. **But**, what if somewhere down the line we wanted to redefine what it means to buy something? Then we would have to change the original class' buy() method. But what if after that, you wanted to redefine what it means to sell something? Then you would have to change the original class' sell() method- **wait a minute that's one too many reasons to change!**

The point being made is that this class does in fact have more than one responsibility as we have just discovered 2 cases in which we need to change the rigid code!!

But how do we fix it? Luckily, SRP is a pretty quick fix. All we have to do is refactor the code in a way such that the Transaction class can still perform two separate tasks but is no longer responsible for their implementation which would ultimately give it one responsibility which would be to perform the Transaction behaviour.. So, in our specific case, our new refactored code looks like:

```
class Transaction{
   private void Buy(String stock, int quantity, float price){
       Buy.buySomething(stock, quantity, price);
   }
   private void Sell(String stock, int quantity, float price){
       Sell.sellSomething(stock, quantity, price);
   }
}
class Buy{
   static void buySomething(String ticker, int quantity, float price){
   }
}
class Sell{
   static void sellSomething(String ticker, int quantity, float price){
   }
}
```

So how exactly have we refactored the code? Well, let's go back to our scenario. If I wanted to redefine what it means to buy something, I wouldn't have to change Transaction. And alternatively, if I wanted to redefine what it means to sell something, I still wouldn't have to change Transaction (so far no reasons to change Transaction). Thus, our code officially conforms to the Single Responsibility Principle!

By Gabriel Vainer and Juefei Lu

Open/Closed Principle: *Software entities should be open for extension, but closed for modification*
Design Smell: *rigidity, repetition,repetition, repetition, repetition, repetition*

Why do we have interfaces or abstract classes? Sure, it can group things together, but its role in software falls into the subject of abstraction, which is the main focal point of why the O/CP is needed in the first place. This type of approach allows extended functionality via concrete implementation classes without needing to change our original classes. Moreover, when implemented correctly we can add new functionality without changing existing code!

Let's go back to our Transaction example, but we'll add some extra functionality. This time, we are going to add a feature that records a transcript of every time the client decides to make a transaction which we will denote as the method makeRecord():

```
class Transaction {
    private void buy(String ticker, int quantity, float price){
        Buy.execute(ticker, quantity, price);
    }
    private void sell(String ticker, int quantity, float price){
        Sell.execute(ticker, quantity, price);
    }
}
class Buy{
    public static String makeRecord(String ticker, int quantity, float price, String date){
        return MessageFormat.format("{0}-{1}-{2}-{3}",
                ticker, String.valueOf(quantity),String.valueOf(price),date);
    }
    static void execute(String stock, int quantity, float price){
    }
}
class Sell{
    public static String makeRecord(String ticker, int quantity, float price, String date){
        return MessageFormat.format("{0}-{1}-{2}-{3}",
                ticker, String.valueOf(quantity),String.valueOf(price),date);
    }
    static void execute(String stock, int quantity, float price){
    }
}
```

wait a minute do you guys see that?? The code is repeating itself! Because makeRecord has the exact implementation for every class it appears in, it shouldn't be open for modification for every class it appears in as well!

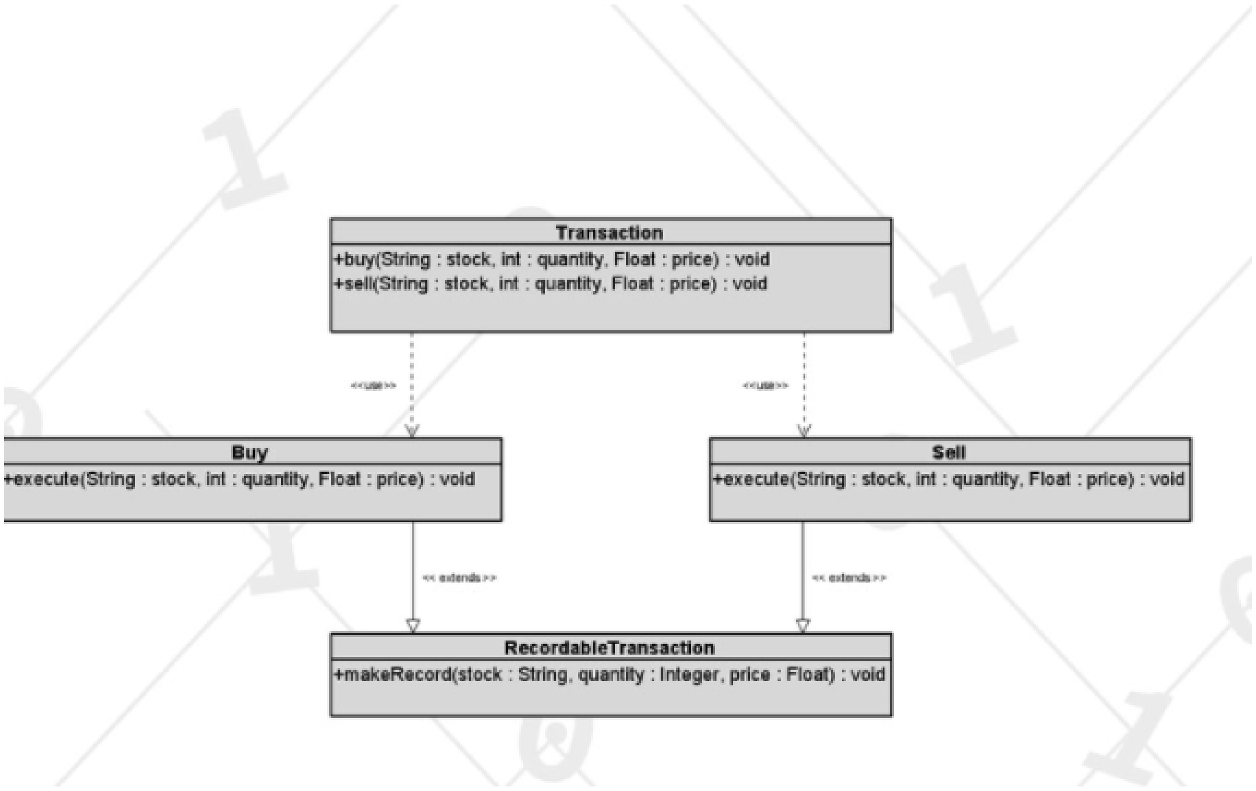
To further get the picture, if at any point we want to redefine what makeRecord is doing, we would essentially change them for both Buy and Sell, when we can save ourselves the trouble and use an abstract class to only switch it in one place!

As displayed below, the Buy and Sell classes, now children of the RecordableTransaction class, both share the makeRecord method and its implementation.

```
abstract class RecordableTransaction {
   public static String makeRecord(String ticker, int quantity, float price, String date){
       return MessageFormat.format("{0}-{1}-{2}-{3}",
               ticker, String.valueOf(quantity),String.valueOf(price),date);
   }
}
```

By Gabriel Vainer and Juefei Lu

```java
class Transaction {
    private void buy(String ticker, int quantity, float price){
        Buy.execute(ticker, quantity, price);
    }
    private void sell(String ticker, int quantity, float price){
        Sell.execute(ticker, quantity, price);
    }
}
class Buy extends RecordableTransaction{
    static void execute(String stock, int quantity, float price){
    }
}
class Sell extends RecordableTransaction{
    static void execute(String stock, int quantity, float price){
    }
}
```



Transaction
+buy(String : stock, int : quantity, Float : price) : void
+sell(String : stock, int : quantity, Float : price) : void
<<use>>
<<use>>
Buy
+execute(String : stock, int : quantity, Float : price) : void
Sell
+execute(String : stock, int : quantity, Float : price) : void
<< extends >>
<< extends >>
RecordableTransaction
+makeRecord(stock : String, quantity : Integer, price : Float) : void

By Gabriel Vainer and Juefei Lu

Liskov Substitution Principle:*If it looks like a duck, quacks like a duck, but needs Batteries you probably have the wrong abstraction*
Design Smell: *fragility, immobility,reliability, unnecessary complexity*

What exactly does it mean for X to be a subset of Y in this context? Well,

- "X is a subtype of Y" means
    - If X is an interface, then Y is also an interface, and X extends Y
    - If X is a class, then
        - If Y is an interface, then X implements Y
        - If Y is a class, then X extends Y

via https://gist.github.com/ThatsJustCheesy/f1cbe7f49ca3d81793e9da581f9e35b1, another great reference for SOLID notes (Shoutout to Ian Gregory).

So what do subtypes and supertypes have to do with LSP? According to Liskov, the relationship [of how subtypes and supertypes] should ensure that any property proved about supertype objects also holds for its subtype objects.

In simple words, this basically means that the subtype should have the same capabilities and incorporate the same behaviour as its superclass. Let's take a closer look as to what this actually means using our reliable Transaction example. This time, we are going to have a new type of Transaction which extends from Transaction. We will also let Transaction be an abstract class. Instead of showing a violation first, I will demonstrate the correct way and then explain how the scenarios that will violate LSP:

```
class Transaction{
    public void buy(String stock, int quantity, float price){
    };
    public void sell(String stock, int quantity, float price){
    };
}
class StockTransaction extends Transaction{
    @Override
    public void buy(String stock, int quantity, float price){
    }
    @Override
    public void sell(String stock, int quantity, float price){
    }
}
```

As seen above, this shows a standard way of dealing with subtypes and effective abstraction. The approach conforms to the Liskov Substitution Principle as substituting a subclass instance of StockTransaction for a superclass instance of Transaction will not break the core functionality of the design. However if we gave StockTransaction an extra method that isn't in Transaction or somehow made StockTransaction to be designed in a way such that it doesn't include all of Transactions' behaviour, this would be considered a violation of the LSP.

Links for more great examples i found:

- https://www.baeldung.com/java-liskov-substitution-principle
- https://stackoverflow.com/questions/56860/what-is-an-example-of-the-liskov-substitution-principle

Interface Segregation Principle: *"Mr Gorbachev, tear down this interface!!" - Ronald Reagan during hackathon in Berlin (Gorbachev violated SOLID)*
Design Smell: *unnecessary complexity, fragility*

This one is relatively simple, stop writing big interfaces dummy. This might be sufficient at the time of making it but it's an incredibly unhealthy practice to have different classes incorporate things they don't even need. Let's do a quick, new, example for an athlete:

```
public interface Athlete {
    void compete();
    void swim();
    void highJump();
    void longJump();
}
```

Simple enough. Just an interface with some methods, and assuming that every subtype of this class includes all of these methods, we should be fine. However, the problem with this is that there is a behaviour difference between swimming and doing high jumps and low jumps, as we will find out with our client, who is primarily a swimmer:

```
public class Phelps implements Athlete {
    @Override
    public void compete() {System.out.println("Phelps started competing")}
    @Override
    public void swim() {System.out.println("Phelps started swimming");}
    @Override
    public void highJump() {}
    @Override
    public void longJump() {}
}
```

Notice how, for the context of a swimmer, we have nothing to put for the jumps because they are two separate types of sports? This is the problem. By implementing the Athlete interface, we have to implement methods like highJump and longJump, which Phelps will never use. The same problem will occur for another athlete who might be a field Athlete competing in the high jump and long jump. They will never implement anything swim related!

The answer should be clear at this point. We will create two other interfaces — one for Jumping athletes and one for Swimming athletes. And thus, a swimmer or a jumper client won't have to use implementations they don't need! Let's observe:

```
public interface Athlete {void compete();}
public interface JumpingAthlete extends Athlete {void swim();}
public interface JumpingAthlete extends Athlete {
    void highJump();
    void longJump();
}
public class Phelps implements SwimmingAthlete {
    @Override
    public void compete() {System.out.println("John Doe started competing");}
    @Override
    public void swim() {System.out.println("John Doe started swimming");}
    }
```

By Gabriel Vainer and Juefei Lu

Dependency Inversion Principle: *High-level modules should not depend on low-level modules. Both should depend on abstractions.Abstractions should not depend on details. Details should depend on abstractions*
Design Smell: *opacity, viscosity,rigidity*

DIP approaches good design in a twofold fashion. Firstly, it dictates that high-level packages/modules shouldn't depend on low-level modules but that both should depend on abstractions. Secondly, it states that details of software constructs should depend on abstractions, not the other way around.

An easier way to think about it is when you have a class with a concrete user class(a low-level module) as a parameter, also known as a high level module. We don't want that because it immediately makes the code super rigid! Lets observe using a library example.

```
public class Book {
    void seeReviews() {...}
    void readSample() {...}
}
public class Shelf {
     Book book;
     void addBook(Book book) {...}
     void customizeShelf() {...}
}
```

From first glance, what's wrong? Objectively…..nothing, this is why DIP can be a bit tricky since we're putting more focus on the behaviour than the code. However, notice how Shelf is depending on a concrete class which violates DIP heavily. But why is this such a bad thing and why does DIP seem so redundant? This can simply be answered with our usual route.. Lets break the code :D:

The main reason why DIP is so crucial is that it's a gateway to violating more principles. Let's say we wanted to add a DVD class which works the exact same as a book. Now it becomes way more obvious that Shelf would now have to adapt for both accepting a book **and** a dvd! Not to mention that this is also a violation of the O/C principle since we have a ton of repetition!!

In order to fix it, we need to make sure that our high level module doesn't rely on a low-level, just as we mentioned. Not only would this fix our code to adapt for dvd's, but this would also get rid of repeated code! It is safe to say that conforming to DIP is also conforming to O/C (that's what I call a 2 for 1).

```
public class Book implements Product {
    @Override
    public void seeReviews() {...}
    @Override
    public void getSample() {...}
}
public class DVD implements Product {
    @Override
    public void seeReviews() {...}
    @Override
    public void getSample() {...}
} (continued on next page)
```

Thus, our shelf now looks cleaner and nice!:

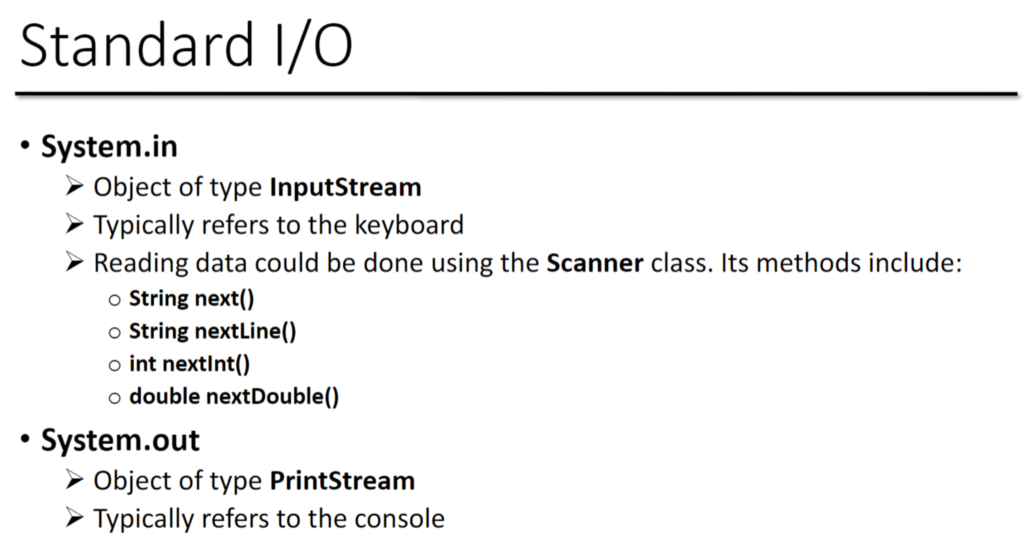
```
public class Shelf {
    Product product;
    void addProduct(Product product) {...}
    void customizeShelf() {...}
```

By Gabriel Vainer and Juefei Lu

```
}
```

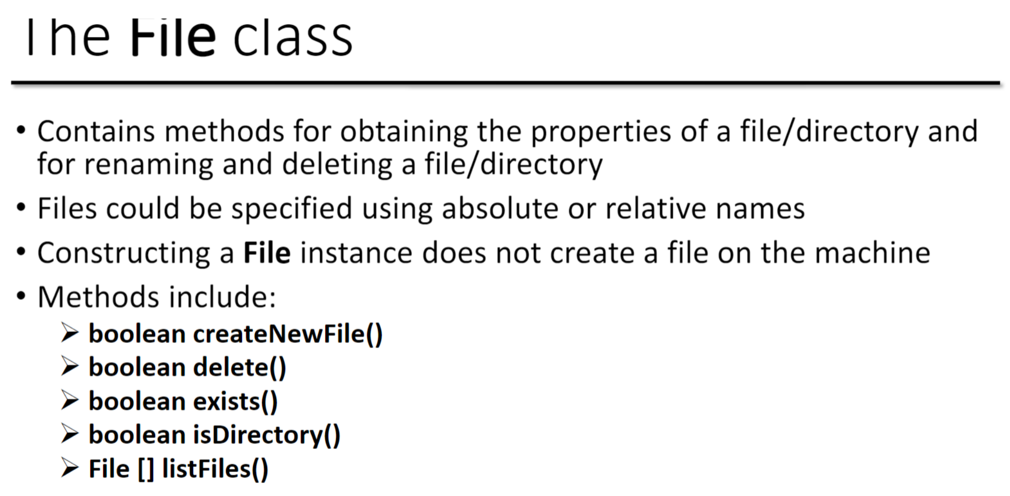
IO Stuff

By Gabriel Vainer and Juefei Lu



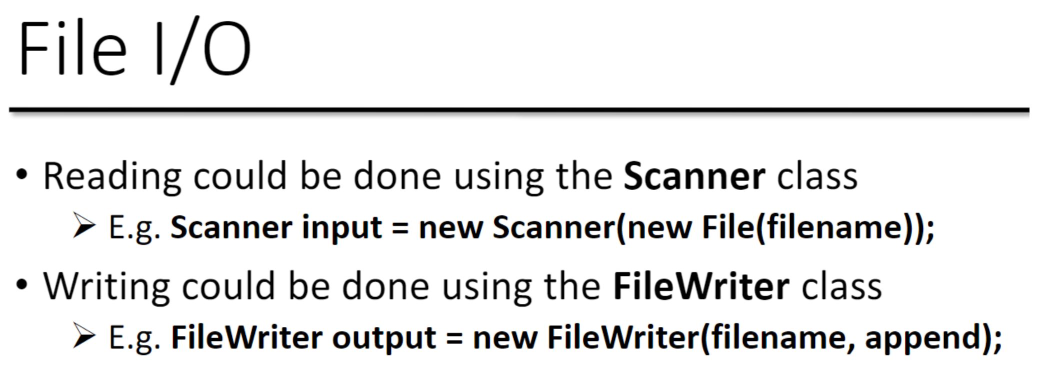
Standard I/O

- **System.in**
  - Object of type **InputStream**
  - Typically refers to the keyboard
  - Reading data could be done using the **Scanner** class. Its methods include:
    - **String next()**
    - **String nextLine()**
    - **int nextInt()**
    - **double nextDouble()**
- **System.out**
  - Object of type **PrintStream**
  - Typically refers to the console



The **File** class

- Contains methods for obtaining the properties of a file/directory and for renaming and deleting a file/directory
- Files could be specified using absolute or relative names
- Constructing a **File** instance does not create a file on the machine
- Methods include:
  - **boolean createNewFile()**
  - **boolean delete()**
  - **boolean exists()**
  - **boolean isDirectory()**
  - **File [] listFiles()**

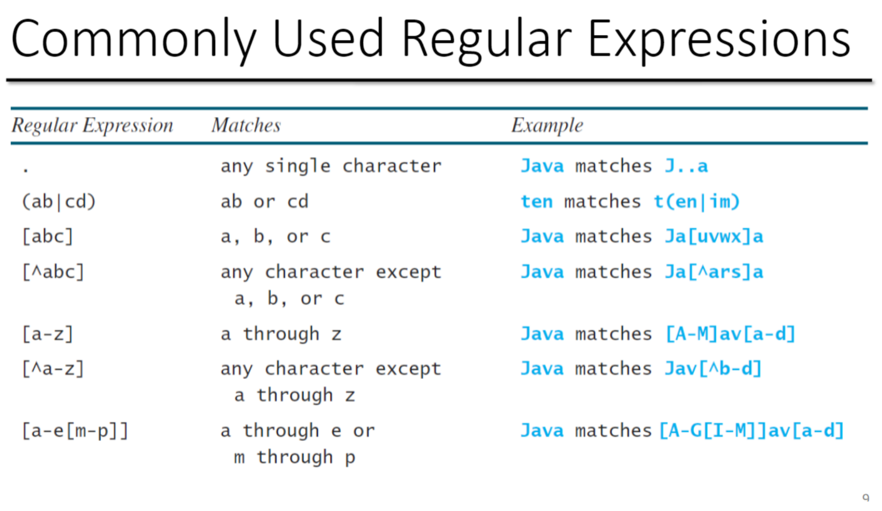


File I/O

- Reading could be done using the **Scanner** class
  - E.g. **Scanner input = new Scanner(new File(filename));**
- Writing could be done using the **FileWriter** class
  - E.g. **FileWriter output = new FileWriter(filename, append);**

IO is short for Input and output and using the methods above you can choose to output to either standard output, which is the terminal or direct it to files, same for inputs.

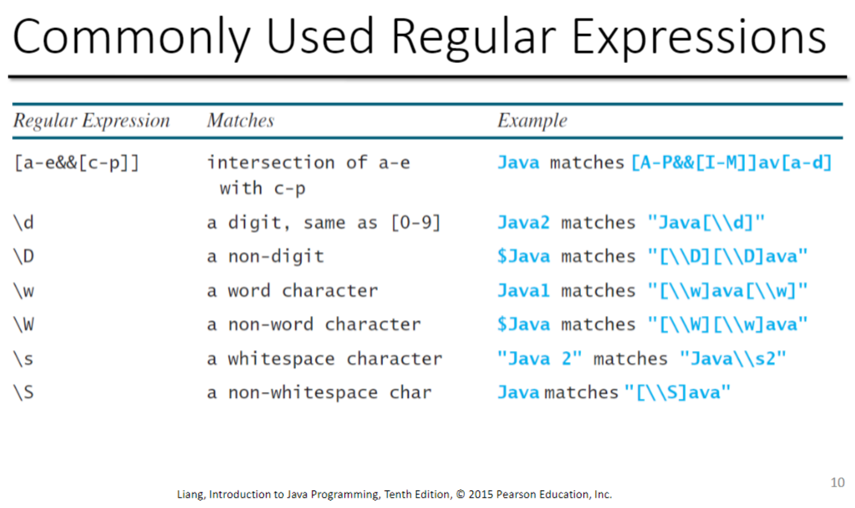
Java Regex



Commonly Used Regular Expressions

Regular Expression	Matches	Example
.	any single character	Java matches J..a
(ab|cd)	ab or cd	ten matches t(en|im)
[abc]	a, b, or c	Java matches Ja[uvwx]a
[^abc]	any character except a, b, or c	Java matches Ja[^ars]a
[a-z]	a through z	Java matches [A-M]av[a-d]
[^a-z]	any character except a through z	Java matches Jav[^b-d]
[a-e[m-p]]	a through e or m through p	Java matches [A-G[I-M]]av[a-d]

9

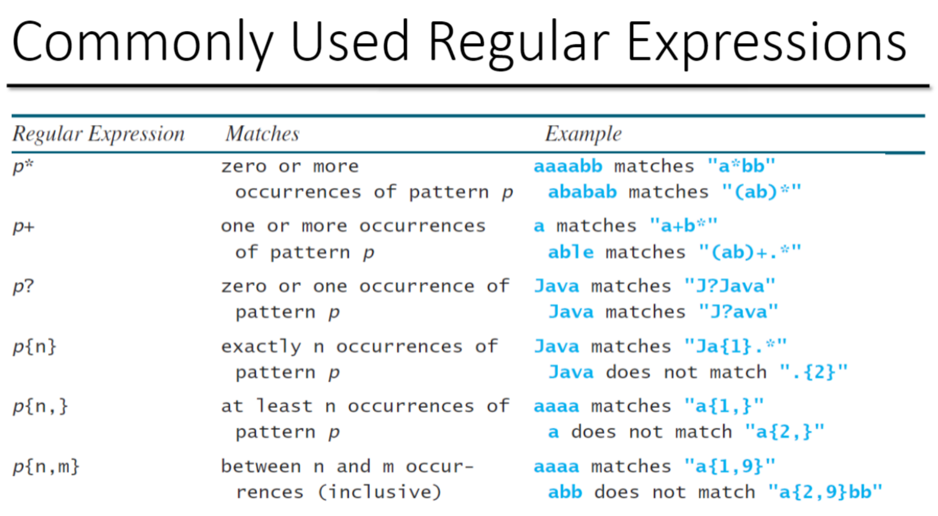


Commonly Used Regular Expressions

Regular Expression	Matches	Example
[a-e&&[c-p]]	intersection of a-e with c-p	Java matches [A-P&&[I-M]]av[a-d]
\d	a digit, same as [0-9]	Java2 matches "Java[\\d]"
\D	a non-digit	$Java matches "[\\D][\\D]ava"
\w	a word character	Java1 matches "[\\w]ava[\\w]"
\W	a non-word character	$Java matches "[\\W][\\w]ava"
\s	a whitespace character	"Java 2" matches "Java\\s2"
\S	a non-whitespace char	Java matches "[\\S]ava"

Liang, Introduction to Java Programming, Tenth Edition, © 2015 Pearson Education, Inc.

10



Commonly Used Regular Expressions

Regular Expression	Matches	Example
*p**	zero or more occurrences of pattern *p*	aaaabb matches "a*bb" ababab matches "(ab)*"
p+	one or more occurrences of pattern *p*	a matches "a+b*" able matches "(ab)+.*"
p?	zero or one occurrence of pattern *p*	Java matches "J?Java" Java matches "J?ava"
p{n}	exactly n occurrences of pattern *p*	Java matches "Ja{1}.*" Java does not match ".{2}"
p{n,}	at least n occurrences of pattern *p*	aaaa matches "a{1,}" a does not match "a{2,}"
p{n,m}	between n and m occurrences (inclusive)	aaaa matches "a{1,9}" abb does not match "a{2,9}bb"

These are the components of making a regex. These can be pieced together into more complicated structures by combining them in a particular order.

For example: `([01]?[0-9]|2[0-3]):[0-5][0-9]`
So it can start with 1 and pick one number from 0-9 so it goes from 10-19 and also 00-09
Or it starts with 2 and can be 20-23, matched with a semicolon and then 00-59 which are the minutes. So for Regex the important thing is to look at the stuff one at a time and figure out the rules so that you can check whether or not the pattern matches your text.

The example is to match a 24 hour clock and is about as complex as the questions on the final exam can possibly be based on the quiz questions.