Exception Handling with try

This lesson introduces the try control structure and teaches you how it can be used for exception handling.

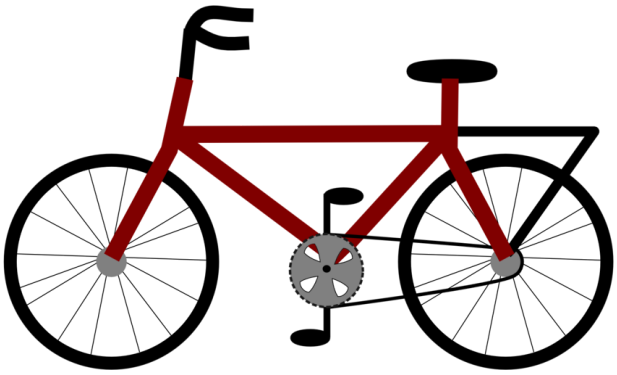
We'll cover the following

Exception Handling

When running a program, we sometimes encounter unexpected conditions which give us errors and warnings. These conditions are known as **exceptions**. While *exceptions* are unexpected, they need to be anticipated by the programmer and only then can they be dealt with in a timely manner. The process of skillfully handling *exceptions* is known as **exception handling**.

Let's take a real-life example to better visualize exception handling.

Imagine you own a courier service which delivers packages to people's homes on bikes. During one such delivery, one of the bikes breaks down unexpectedly. While this was unexpected, you still anticipated bikes to break down and kept a spare as back up. Because you handled the situation, the customer was able to receive their package in a timely manner. This is exception handling; handling *exceptions* in a program to prevent premature termination of the code.



In Scala, we handle exceptions using the `try` control structure. Let's see how it works below.

`try`

`try` is used to *try* particular parts of a code that might throw exceptions. If an exception is thrown, it is caught using `catch`. When the compiler catches an exception, it chooses how to handle it from a list of provided cases.

Let's look at the syntax of `try` below.

Syntax

```
try {
  block of code
} catch {
  case ex: type of exception =>
  case ex: type of exception =>
}
```

The block of code in which you anticipate an exception is wrapped in a `try` expression. This is followed by the `catch` statement which catches the exception and runs different cases until it finds the correct exception. The statement after `=>` tells the compiler what to do in place of executing the block of code.

> Scala reuses Java's exceptions.

Let's look at a very common exception: dividing by zero.

Using `try`

Dividing a number by another number is a simple enough task, but sometimes we end up diving by zero which causes the program to crash and pre-maturely terminate.

This code requires the following environment variables to execute:

LANG C.UTF-8

```
val dividend = 10
val divisor = 0

val quotient = dividend/divisor
println(quotient)
```

Running the above code gives an `ArithmeticException` as it is thrown during an error while doing an arithmetic calculation. How would we handle this using `try`?

This code requires the following environment variables to execute:

LANG C.UTF-8

```
val dividend = 10
val divisor = 0

try {
  val quotient = dividend/divisor
  println(quotient)
}
catch {
  case ex: ArithmeticException => println("Dividing by zero is not allowed")
}
```

The code above will print `"Dividing by zero is not allowed"` instead of crashing like it did in the first program. Try changing `divisor` to a non-zero value and see what happens.

Let's move on to our final control structure: `match`.