Naming Things in Code

1. Don't Abbreviate Names

Negative example:

In [84]:
```
data class Mov(val gen: String, val yr: Int, val dir: String, val writers: List<

fun relScore(m1: Mov, m2: Mov): Double {
    val GW = 0.4
    val YW = 0.1
    val DW = 0.2
    val WW = 0.3

    var s = 0.0
    if (m1.gen == m2.gen) {
        s += GW
    }

    if (m1.yr == m2.yr) {
        s += YW
    }

    if (m1.dir == m2.dir) {
        s += DW
    }

    if (m1.writers.any(m2.writers::contains)) {
        s += WW
    }

    return s;
}
```

Positive example:

In [85]:
```
data class Movie(val genre: String, val year: Int, val director: String, val wri

private val GENRE_WEIGHT = 0.4
val YEAR_WEIGHT = 0.1
val DIRECTOR_WEIGHT = 0.2
val WRITER_WEIGHT = 0.3

fun movieRelationScore(movie1: Movie, movie2: Movie): Double {

    var score = 0.0
    if (movie1.genre == movie2.genre) {
        score += GENRE_WEIGHT
    }

    if (movie1.year == movie2.year) {
        score += YEAR_WEIGHT
    }
```

```
    if (movie1.director == movie2.director) {
        score += DIRECTOR_WEIGHT
    }

    if (movie1.writers.any(movie2.writers::contains)) {
        score += WRITER_WEIGHT
    }

    return score;
}
```

Real Examples:

```
    @Reference
    private ConfigurationAdmin cm;

    private ImmutableList<EnergyScheduleHandler> eshs;

    .setSimulator((id, period, gsc, coc, csc, ef, mode, fitness,
    isFinalRun) -> { ... }
```

1.1 Use Pronouncable Names

In [86]:
```
import java.util.Date

val genymdhms: Date = Date() // generation year, month, day, hour, minute and se
```

In [87]:
```
val generationTimestamp: Date = Date()
```

2. Make Meaningful Distinctions

`getActiveAccount()` vs. `getActiveAccounts` vs. `getActiveAccountInfo()`

`moneyAmount` vs. `money`

`customerInfo` vs. `customer`

`accountData` vs. `account`

3. Add Units to Variables Unless the Type Tells You

In [88]:
```
import kotlin.time.Duration

fun execute1(delay: Int) {}

// Better
fun execute2(delaySeconds: Int) {}

// Best
fun execute3(delay: Duration) {}

execute3(30.seconds)
```

Real Examples:

```
var soc // --> int soc, var socPercentage

var gridBuyEnergy // --> var gridBuyEnergyWh
```

4. Don't Put Types in Your Types

- Prefixes like `I`, `Base`, or `Abstract` leak implementation details into names
- Users don't care whether something is an interface or abstract class, only what it represents
- If you struggle to name a superclass, it's often a sign that the structure is wrong or the subclass has the wrong responsibility

In [89]:
```
interface Movable {}
// abstract class Movable {}
// class Movable {}
```

In [90]:
```
class PositionAnimation(movable: Movable) {}
```

In [91]:
```
// Don't
interface IMovable {}
abstract class BaseMovable {}
abstract class AbstractMovable {}
```

5. Refactor If You Find Yourself Naming Code "Utils"

- **No clear responsibility:** Everything gets dumped into one place, even if it doesn't belong together
- **Hides the domain model:** Behavior is removed from real objects and placed in a neutral helper class
- **Low cohesion:** Methods in Utils often have no real relation to each other

a) Move Methods to Their Respective Types

In [92]:
```
// Don't
object Utils {
    fun assignDefaults(movie: Movie) {}

    fun relationScore(movie1: Movie, movie2: Movie): Double {
        return 0.0
    }
}

// Do
class Movie {
    fun assignDefauls() {}

    fun relationScore(other: Movie): Double {
        return 0.0
    }
}
```

b) Replace Utility Classes with Domain Objects

In [93]:
```
data class Movie(val rating: Double)

// Don't
class Utils {
    fun topMoviesByRating(movies: List<Movie>, numTop: Int): List<Movie> {
        return movies.sortedByDescending { it.rating }.take(numTop)
    }
}

// Do
class MovieCollection(
    private val movies: List<Movie>
) {
    fun topByRating(numTop: Int): List<Movie> =
        this.movies.sortedByDescending { it.rating }.take(numTop)
}


fun List<Movie>.topByRating(numTop: Int = 10): List<Movie> =
    this.sortedByDescending { it.rating }.take(numTop)
```