Optional

The concept & Best practice

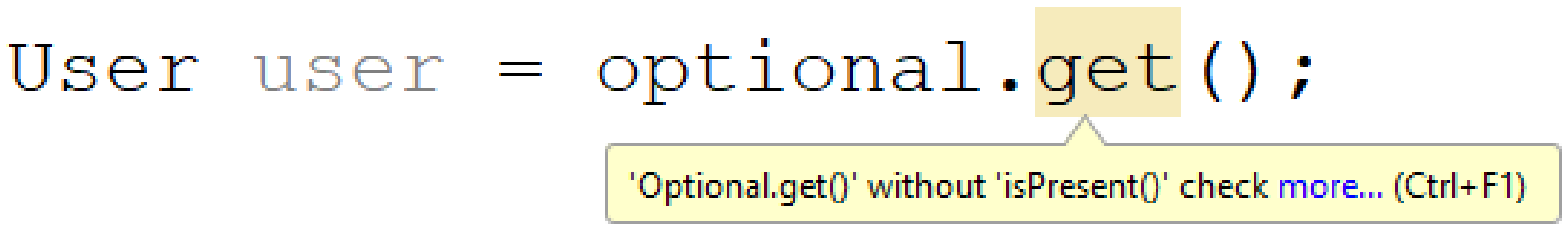
NULL POINTER EXCEPTIONS
NULL POINTER EXCEPTIONS
EVERYWHERE
quickmeme.com

The concept

- is a polymorphic type that represents encapsulation of an optional value; e.g., it is used as the return type of functions which may or may not return a meaningful value when they are applied. It consists of a constructor which either is empty (named *None* or *Nothing*), or which encapsulates the original data type A

Rule 1 – never use get()

Available, shortest and simplest method, which misses the goal



```
User user = optional.get();
```

'Optional.get()' without 'isPresent()' check more... (Ctrl+F1)

```
T get() {
    if (value == null) {
        throw new NoSuchElementException("No value present");
    }
    return value;
}
```

No such element instead of NPE

Rule 2 – you don’t need isPresent()

```
@Service
@Transactional
public class UglyService {
    @Autowired
    private UserRepository repository;

    public String getName(Long userId) {
        Optional<User> optional = repository.findById(userId);
        if (optional.isPresent()) {
            return optional.get().getUserName();
        }else {
            return "Alex";
        }
    }
}
```

Procedural style vs functional

```
public String getName(Long userId) {
    Optional<User> optional = repository.findById(userId);
    if (optional.isPresent()) {
        return optional.get().getUserName();
    }else {
        return defaultUser.getUserName();
    }
}


public String getName(Long userId) {
    return repository.findById(userId).orElse(defaultUser).getUserName();
}
```

Lazy calculation of default value

```
public String getNotLazy(Long userId) {
    return repository.findById(userId)
            .orElse(UserUtils.getDefaultUser())
            .getUserName();
}
```

Evaluated when logical chain will be build

```
public String getNameLazy(Long userId) {
    return repository.findById(userId)
            .orElseGet(() -> UserUtils.getDefaultUser())
            .getUserName();
}
```

Evaluated only when optional is empty

```
public String getNameLazy(Long userId) {
    return repository.findById(userId)
            .orElseGet(UserUtils::getDefaultUser)
            .getUserName();
}
```

Handling exception

```java
public String getName(Long userId) {
    Optional<User> optional = repository.findById(userId);
    if (optional.isPresent()) {
        return optional.get().getUserName();
    }else {
        throw new IllegalStateException(userId + " not exists");
    }
}

public String getName(Long userId) {
    return repository.findById(userId).orElseThrow(() -> {
        throw new IllegalStateException(userId + " not exists");
    }).getUserName();
}
```

Do something if not null

```
public void deleteUser(Long userId) {
    Optional<User> optional = repository.findById(userId);
    if (optional.isPresent()) {
        repository.delete(optional.get());
    }
}
```

```
public void deleteUser(Long userId) {
    repository.findById(userId).ifPresent(user -> repository.delete(user));
}
```

More optional options

```java
public String getName(Long userId) {
   return repository.findById(userId)
           .map(user -> user.getUserName().toUpperCase())
           .orElse("ALEX");
}
```

Creating an optional

`Optional.empty();`

`Optional.of(user);` → **Exception, if user will be null**

`Optional.ofNullable(user);` → **Empty optional, if user will be null**

Patterns with optional

Old style

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Person {
    private Car car;
}
```

```java
@Data
@AllArgsConstructor
public class Car {
    private Insurance insurance;
}
```

Old school

```
public String getInsuranceNameOldStyle(Person person) {
    String name = "no name";
    if (person != null) {
        Car car = person.getCar();
        if (car != null) {
            Insurance insurance = car.getInsurance();
            if (insurance != null) {
                name = insurance.getName();
            }
        }
    }
    return name;
}
```

Moving to Optional

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Person2 {
    private Car2 car;

    public Optional<Car2> getCar() {
        return Optional.ofNullable(car);
    }
}
```

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Car2 {
    private Insurance insurance;

    public Optional<Insurance> getInsurance() {
        return Optional.ofNullable(insurance);
    }
}
```

```java
public String getInsuranceNameWithOptionalStillOldStyle(Person2 pe
    String name = "no name";
    if (person != null) {
        Optional<Car2> optionalCar = person.getCar();
        if (optionalCar.isPresent()) {
            Car2 car = optionalCar.get();
            Optional<Insurance> insuranceOptional = car.getInsuran
            if (insuranceOptional.isPresent()) {
                name = insuranceOptional.get().getName();
            }
        }
    }
    return name;
}
```

```java
public String getInsuranceNameWithoutLocalVariables(Person2 person2)
    String name = "no name";
    if (Optional.ofNullable(person2).isPresent()) {
        if (person2.getCar().isPresent()) {
            if (person2.getCar().get().getInsurance().isPresent()) {
                name = person2.getCar().get()
                .getInsurance().get().getName();
            }
        }
    }
    return name;
}
```

ifPresent

```java
public String getInsuranceNameWithDirtyHack(Person2 person2) {
    String name = "no name";
    Optional.ofNullable(person2).ifPresent(
            p->p.getCar().ifPresent(
                    c->c.getInsurance().ifPresent(
                            n-> name=n;
                    )
            )
    );
    return name;
}
```

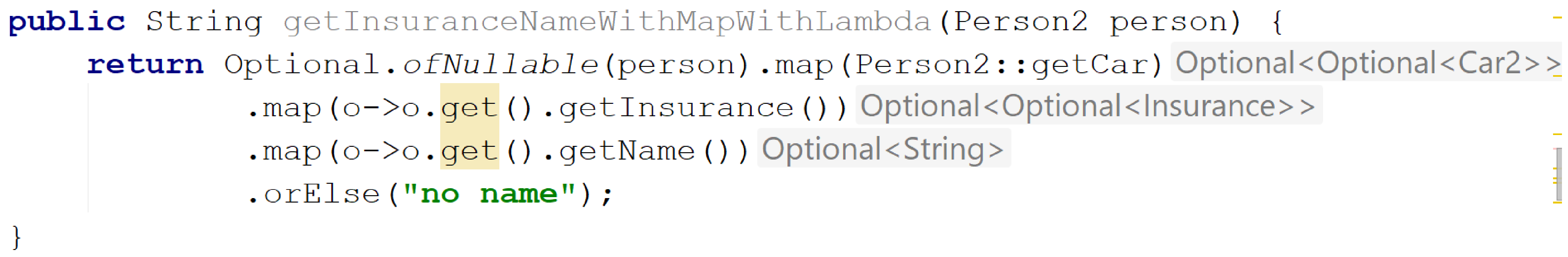
Compilation error (name – effectively final)

```
public String getInsuranceNameWithDirtyHack(Person2 person2) {
    String name = "no name";
    Optional.ofNullable(person2).ifPresent(
            p->p.getCar().ifPresent(
                    c->c.getInsurance().ifPresent(
                            n-> name=n;
                    )
            )
    );
    return name;
}
```

We know all dirty hacks

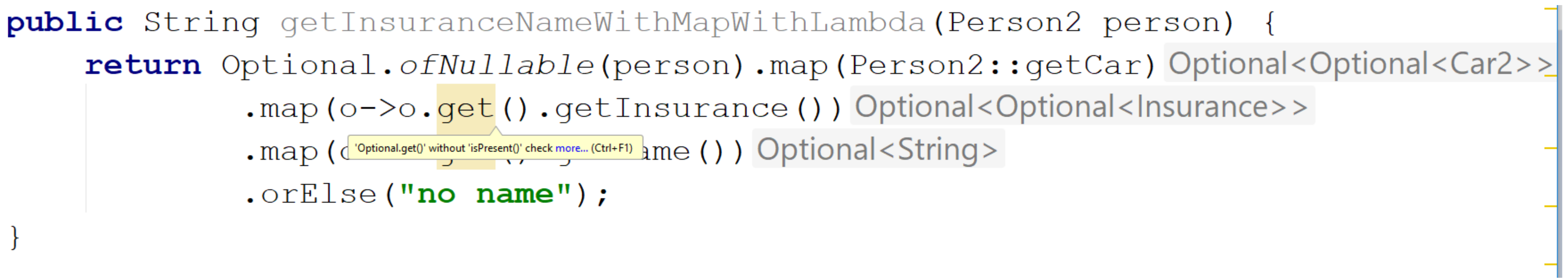
```java
public String getInsuranceNameWithMap(Person2 person) {
    return Optional.ofNullable(person).map(Person2::getCar)
            .map(Optional::get).map(Car2::getInsurance)
            .map(Optional::get).map(Insurance::getName)
            .orElse("no name");
}
```

In one lambda doesn’t make the difference



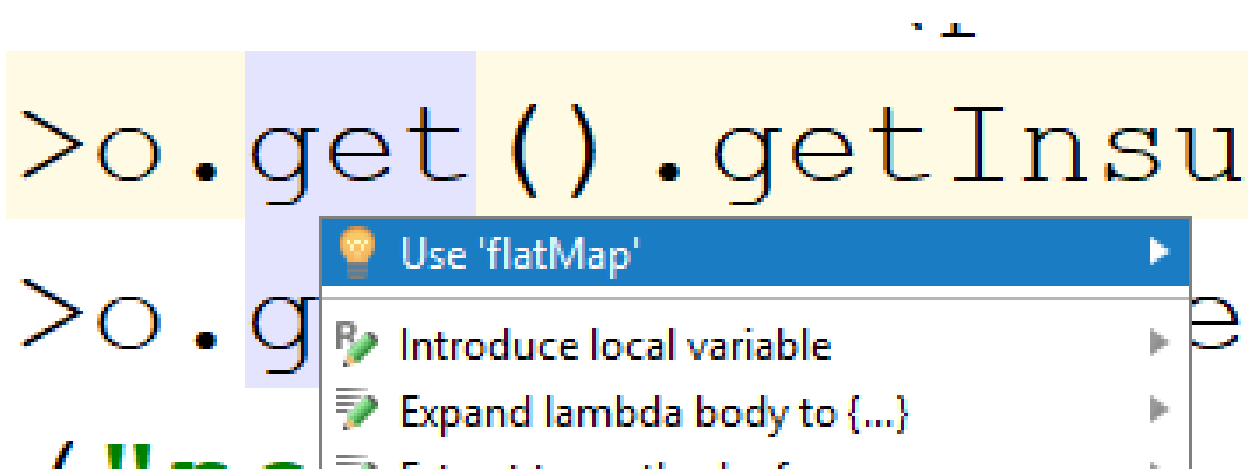
```
public String getInsuranceNameWithMapWithLambda(Person2 person) {
    return Optional.ofNullable(person).map(Person2::getCar) Optional<Optional<Car2>>
                .map(o->o.get().getInsurance()) Optional<Optional<Insurance>>
                .map(o->o.get().getName()) Optional<String>
                .orElse("no name");
}
```

Intellij recommends…



```
public String getInsuranceNameWithMapWithLambda(Person2 person) {
    return Optional.ofNullable(person).map(Person2::getCar) Optional<Optional<Car2>>
            .map(o->o.get().getInsurance()) Optional<Optional<Insurance>>
            .map(c 'Optional.get()' without 'isPresent()' check more... (Ctrl+F1) ame()) Optional<String>
            .orElse("no name");
}
```

Intellij solution



```
>o.get().getInsu
>o.g
```

optional.flatMap

```java
public String getInsuranceNameWithFlatMap(Person2 person) {
    return Optional.ofNullable(person)
            .flatMap(Person2::getCar)
            .flatMap(Car2::getInsurance)
            .map(Insurance::getName)
            .orElse("no name");
}
```

```java
public <U> Optional<U> map(Function mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent()) {
        return empty();
    } else {
        return Optional.ofNullable(mapper.apply(value));
    }
}
```

map

```java
public <U> Optional<U> flatMap(Function mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent()) {
        return empty();
    } else {
        @SuppressWarnings("unchecked")
        Optional<U> r = (Optional<U>) mapper.apply(value);
        return Objects.requireNonNull(r);
    }
}
```

flatMap

Optional map/flatMap conclusion

- When maping optional value
  - Use map if your method return not optional
  - Use flatMap if your method return optional

```
Optional<Insurance> getInsurance()
```

```
.flatMap(Car2::getInsurance) Optional<Insurance>
.map(Insurance::getName) Optional<String>
```

```
String getName()
```

```
    String name = "no name";
        if (person != null) {
            Car car = person.getCar();
            if (car != null) {
                Insurance insurance = car.getInsurance();
                if (insurance != null) {
                    name = insurance.getName();
                }
            }
        }
        return name;
    }
```

Java 7

```
        return Optional.ofNullable(person)
                .flatMap(Person2::getCar)
                .flatMap(Car2::getInsurance)
                .map(Insurance::getName)
                .orElse("no name");
    }
```

Java 8 correct

```
    person?.car?.insurance?.name?:"no name"
```

Groovy