Inheritance and Abstract Classes – Part 1

`Lamp` class Example

```
class Lamp(private var isOn: Boolean) {
    fun pressSwitch() {
        isOn = !isOn
    }

    override fun toString(): String =
        if (isOn) {
            "LIGHT"
        } else {
            "(darkness)"
        }
}
```

```
fun main() {
    val lamp = Lamp(false)
    lamp.show()
    lamp.pressSwitch()
    lamp.show()
}
```

Output:

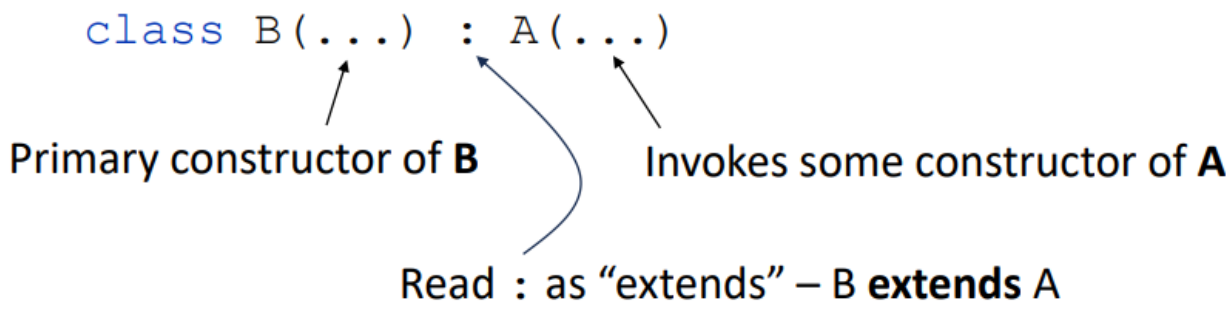
```
(darkness)
LIGHT
```

```
class DimmingLamp(
    isOn: Boolean,
) : Lamp(isOn) {
    private var brightness: Int =
        if (isOn) { 10 } else { 0 }

    override fun pressSwitch() {
        super.pressSwitch()
        if (isOn) {
            brightness = 10
        } else {
            brightness = 0
        }
    }

    fun up(): DimmingLamp {
        if (isOn && brightness < 10) {
            brightness++
        }
        return this
    }

    fun down(): DimmingLamp {
        if (isOn && brightness > 1) {
            brightness--
        }
        return this
    }

    override fun toString(): String =
        super.toString() +
            if (isOn) {
                ": " + "*".repeat(brightness)
            } else {
                ""
            }
}
```

This won't work!

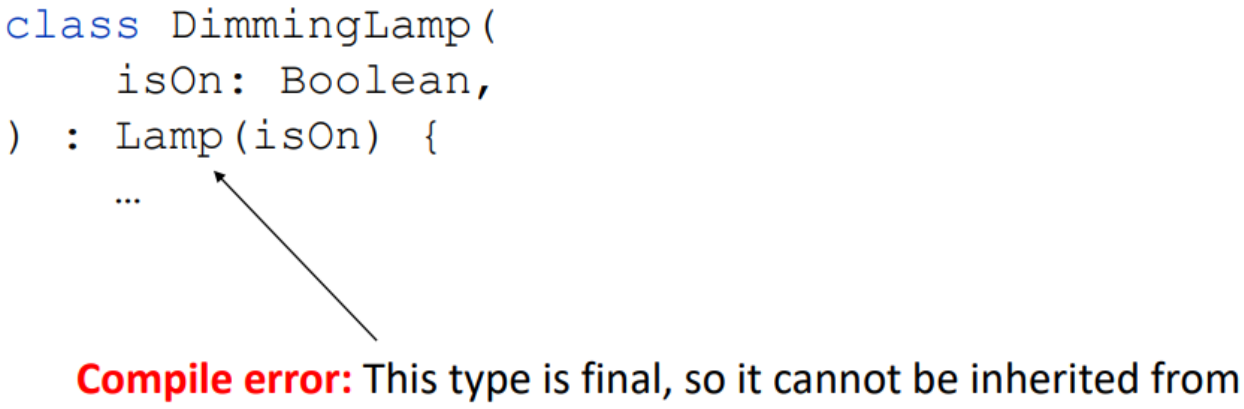
Extending a superclass

To indicate that a class B extends an existing class A, write:

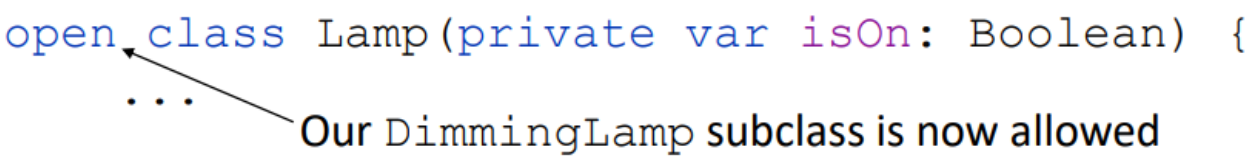


Same syntax as when implementing an interface, except **that we write down the constructor of A**.

Only `open` classes can be extended



`open` indicates that subclasses of a class are allowed



Classes are final (i.e. not open) by default.
You can declare a class as final, but this is redundant.

This is good: inheritance should be used carefully and sparingly.

Overriding methods

If `foo` is a method in superclass A, subclass B may wish to override `foo`:

```
class B(…) : A(…) {
    …
    override fun foo(…) …
        // Extended or replacement behaviour for
        // foo when invoked on a B instance
```

Same syntax as before.

Only `open` methods can be overridden

To allow subclasses of `Lamp` to override `pressSwitch` we must mark it as `open`

```
open class Lamp(private var isOn: Boolean) {
    open fun pressSwitch() {
        isOn = !isOn
    }
    …
```

We can now override `pressSwitch` in `DimmingLamp` subclass

Using `super` to invoke superclass method

```
class DimmingLamp(…) : Lamp(…) {
    …
    override fun pressSwitch() {
        super.pressSwitch()
        …
    }
```

Execute the Lamp version of `pressSwitch`. Execute some extra code specific to `DimmingLamp`.

A subclass cannot directly access private properties and methods of superclasses

A property or method of a class can be:

- public – visible everywhere in the codebase
- private – only visible inside the class
- **protected – only visible inside the class, or in (direct or indirect) subclasses**

```
open class Lamp(protected var isOn: Boolean) {
    ...
```

Better, since Subclasses of Lamp can read `isOn` and `isOn` is not visible except in Lamp and its subclasses. However, Subclasses of Lamp can modify `isOn`.

So, still slightly violates encapsulation.

```
open class Lamp(isOn: Boolean) {
    protected var isOn: Boolean = isOn
        private set
    ...
```

Best solution: `isOn` is no more visible than necessary.

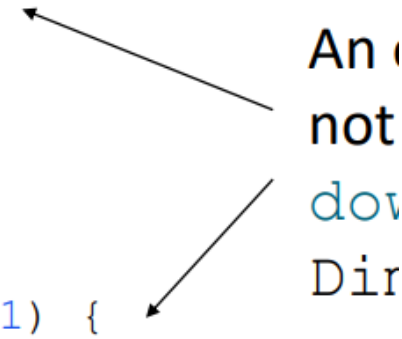
A subclass can add new properties

```
class DimmingLamp(isOn: Boolean) : Lamp(isOn) {
    private var brightness: Int =
```

An ordinary `Lamp` does not have the `brightness` property – only a `DimmingLamp` does

A subclass can add new methods

```
class DimmingLamp(isOn: Boolean) : Lamp(isOn) {
    …
    fun up(): DimmingLamp {
        if (isOn && brightness < 10) {
            brightness++
        }
        return this
    }

    fun down(): DimmingLamp {
        if (isOn && brightness > 1) {
            brightness--
        }
        return this
    }
    …
```



An ordinary `Lamp` does not have the `up` and `down` methods only a `DimmingLamp` does

The complete working Lamp class

```
open class Lamp(isOn: Boolean) {
    protected var isOn: Boolean = isOn
        private set

    open fun pressSwitch() {
        isOn = !isOn
    }

    override fun toString(): String =
        if (isOn) {
            "LIGHT"
        } else {
            "(darkness)"
        }
}
```

Magic Numbers

```
private val MIN_BRIGHTNESS: Int = 1
private val MAX_BRIGHTNESS: Int = 10
```

Avoids duplicate code and magic numbers

Inheritance Terminology

A superclass may also be called:

- Parent class
- Base class

A subclass may also be called:

- Child class
- Derived class

Inheritance is transitive:

- If C is a subclass of B and B is a subclass of A then C is a subclass of A

- **C** is an **indirect subclass** of **A**
- **A** is an **indirect superclass** of **C**
- **C** **indirectly inherits from** **A**

Properties and methods of a class are referred to collectively as members of the class. A subclass inherits the public and protected members of its superclasses

Another Example

```
enum class Terrain {
    WATER, FOREST, SWAMP, ROCKS
}

enum class WorldKind {
    BOUNDED, DEADLY, RANDOM
}

class DeadPlayerException(message: String) : Exception(message)
```

```
class GridWorld(
    private val width: Int,
    private val height: Int,
    private val worldKind: WorldKind,
) {
    private val grid: Array<Array<Terrain>> = randomTerrain()
    private var position: Pair<Int, Int> = randomPosition()

    fun up() = updatePosition(position.copy(second = position.second + 1))

    fun down() = updatePosition(position.copy(second = position.second - 1))

    // left() and right() - similar

    private fun updatePosition(newPosition: Pair<Int, Int>) {
        if (newPosition.first in 0..<width && newPosition.second in 0..<height) {
            position = newPosition
            return
        }
        when (worldKind) {
            WorldKind.BOUNDED -> position = clampToGrid(newPosition)
            WorldKind.DEADLY -> throw DeadPlayerException("Fell of world!")
            WorldKind.RANDOM -> position = randomPosition()
        }
    }
    …
}
```

Exercise: implement these and come
a way of showing the game world as

Problem with this design

Not extensible: the world kinds need to be known upfront.

The `GridWorld` class requires specific knowledge of the world kinds:

```
…
when (worldKind) {
    WorldKind.BOUNDED -> position = clampToGrid(newPosition)
    WorldKind.DEADLY -> throw DeadPlayerException("Fell of world!")
    WorldKind.RANDOM -> position = randomPosition()
}
```

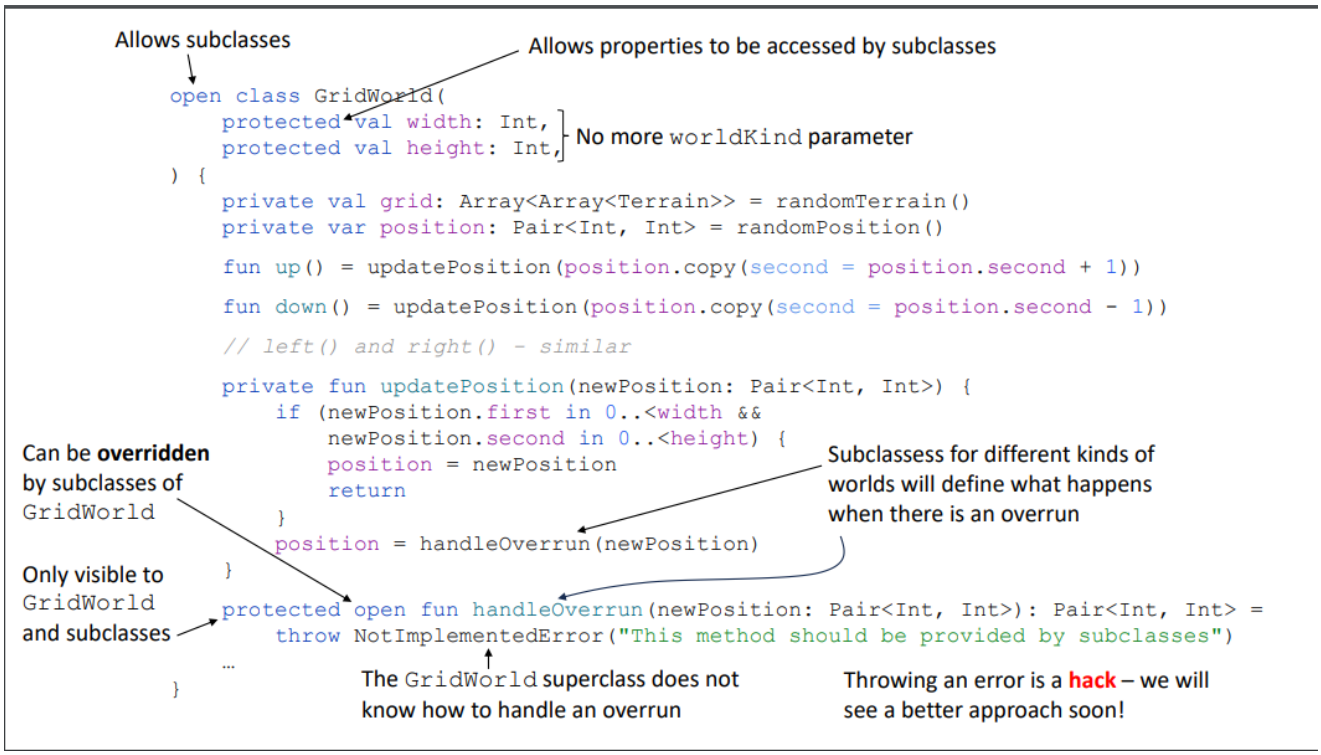
Add a new kind of world:

```
enum class WorldKind {
    BOUNDED, DEADLY, RANDOM, TORUS
}
```

The `GridWorld` class no longer compiles and must be changed.

Alternative design – inheritance

~~enum class WorldKind {~~
~~BOUNDED, DEADLY, RANDOM~~
~~}~~




```
open class GridWorld(
    protected val width: Int,
    protected val height: Int,
) {
    private val grid: Array<Array<Terrain>> = randomTerrain()
    private var position: Pair<Int, Int> = randomPosition()

    fun up() = updatePosition(position.copy(second = position.second + 1))

    fun down() = updatePosition(position.copy(second = position.second - 1))

    // left() and right() - similar

    private fun updatePosition(newPosition: Pair<Int, Int>) {
        if (newPosition.first in 0..<width &&
            newPosition.second in 0..<height) {
            position = newPosition
            return
        }
        position = handleOverrun(newPosition)
    }

    protected open fun handleOverrun(newPosition: Pair<Int, Int>): Pair<Int, Int> =
        throw NotImplementedError("This method should be provided by subclasses")
    ...
}
```

`BoundedGridWorld` subclass

```
class BoundedGridWorld(
    width: Int,
    height: Int,
) : GridWorld(width, height) {
    override fun handleOverrun(newPosition: Pair<Int, Int>): Pair<Int, Int> =
        Pair(
            first = max(0, min(newPosition.first, height - 1)),
            second = max(0, min(newPosition.second, width - 1)),
        )
}
```

`DeadlyGridWorld` subclass

When overriding a method it is OK to narrow the return type

```
class DeadlyGridWorld(
    width: Int,
    height: Int,
) : GridWorld(width, height) {
    override fun handleOverrun(newPosition: Pair<Int, Int>): Nothing =
        throw DeadPlayerException("Fell off world!")
}
```

Nothing – the Kotlin type with no values.
Unit - the Kotlin type with 1 value.

Problems

- We do not want a `GridWorld` object, but nothing stops a client creating one
  - It would be better if we could not create “just a `GridWorld`”
- Nothing forces us to override the dummy superclass method
  - It would be nice if the compiler forced us to implement this method
- The dummy superclass `handleOverrun` implementation is available via `super`
  - Accidental superclass call – leads to exception
  - It would be better if this call was not allowed

Abstract Classes

```
abstract class GridWorld(
    protected val width: Int,
    protected val height: Int,
) {
```

`abstract` before `class` creates an abstract class, means you cannot create direct instances of this class.

An abstract class is automatically open.

An abstract method is automatically open.

`abstract` before `fun` creates an abstract method, it has no default implementation, and concrete subclasses must provide an implementation. Same as for abstract methods of interfaces.

```
protected abstract fun handleOverrun(newPosition: Pair<Int, Int>): Pair<Int, Int>
…
```

Abstract classes can have concrete properties and methods. A concrete method of an abstract class can be defined in terms of abstract methods.

All 3 problems solved.