Inheritance and abstract classes – part 1

Alastair F. Donaldson

Aims of this lecture

- Introduce **inheritance**
- Show an example of writing a **superclass** and a **subclass**
- Introduce **protected** visibility
- Introduce **abstract classes**

More on inheritance and abstract classes in part 2

`Lamp` class

Let’s write a simple class to represent a lamp

The lamp can be turned on and off by pressing a switch

Displaying the lamp as a string indicates whether it is on or off

Internal details of the lamp are completely hidden from clients – **encapsulated** within the class

Lamp class

```
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)
    println(lamp
    lamp.pressSwitch()
    println(lamp
}
```

Output:

```
(darkness)
LIGHT
```

`DimmingLamp` class

A dimming lamp **is a** special kind of lamp

Like a lamp, it has an on/off switch

When on, its brightness can be decreased, and then increased, between a maximum and minimum

Again, `toString()` will show the state of the lamp; all other details of the lamp are **encapsulated**

Let us make `DimmingLamp` a **subclass** of `Lamp`

DimmingLamp subclass

```kotlin
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 {
                ""
            }
}
```

Intended behaviour

```
fun main() {
    val dimmingLamp = DimmingLamp(true)
    println(dimmingLamp)
    dimmingLamp.down().down().down().down().down().down().down()
    println(dimmingLamp)
    dimmingLamp.down().down().down().down().down().down().down()
    println(dimmingLamp)
    dimmingLamp.up().up().up()
    println(dimmingLamp)
    dimmingLamp.pressSwitch()
    println(dimmingLamp)
    dimmingLamp.pressSwitch()
    println(dimmingLamp)
}
```

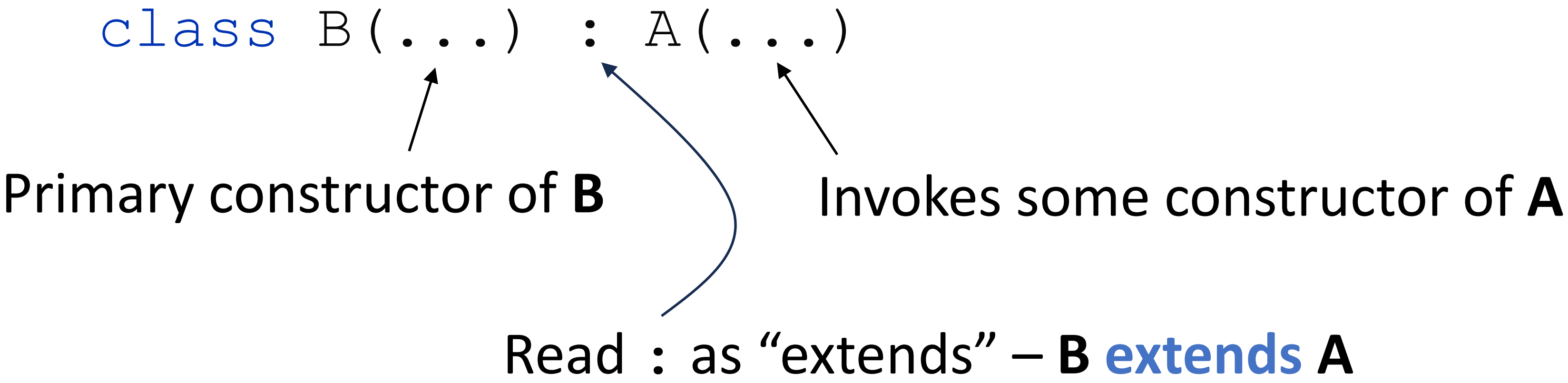
Intended output:

```
LIGHT: **********
LIGHT: ***
LIGHT: *
LIGHT: ****
(darkness)
LIGHT: **********
```

But at present, `DimmingLamp` will not even compile!

Extending a superclass

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



```
class B(...) : A(...)
```

Primary constructor of **B**

Invokes some constructor of **A**

Read **:** as “extends” – **B extends A**

Same syntax as used to say that a class implements an interface, except that we indicate how the constructor of **A** should be invoked

Only **open** classes can be extended

```
class DimmingLamp(
    isOn: Boolean,
) : Lamp(isOn) {
    …
```

Compile error: This type is final, so it cannot be inherited from

To allow subclasses of `Lamp`, we need to mark `Lamp` as `open`

Making `Lamp` an **open** class

`open` indicates that subclasses of a class are allowed

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

Our `DimmingLamp` subclass is now allowed

Classes are **final** (i.e. **not open**) by default

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

Final-by-default means we must actively decide to allow inheritance

You can declare a class as `final`, but this is redundant

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 used to override default methods of interfaces

Only **open** methods can be overridden

```
class DimmingLamp(
    isOn: Boolean,
) : Lamp(isOn) {
    ...

    override fun pressSwitch() {
        ...
    }
}
```

Compile error: 'pressSwitch' in 'Lamp' is final and cannot be overridden

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

Making `pressSwitch` an **open** method

`open` indicates that subclasses can override a method

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

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

Methods are **final** (i.e. **not open**) by default

Again, **final-by-default** is good: overriding is only possible if we actively want to make it possible

You can declare a method as `final`, but this is redundant

Using super to invoke superclass method

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

    override fun toString(): String =
        super.toString() + …
```

Execute the Lamp version of pressSwitch

Execute some extra code specific to DimmingLamp

Get what the string version of the object as a plain Lamp

Add on something DimmingLamp-specific

A subclass cannot directly access private properties and methods of superclasses

```
class DimmingLamp(…) : Lamp(…) {
    …
    override fun pressSwitch() {
        super.pressSwitch()
        if (isOn) {
            brightness = 10
        } else {
            brightness = 0
        }
    }
    …
```

Compile error: Cannot access 'isOn': it is invisible (private in a supertype) in 'DimmingLamp'

Bad solution: make `isOn` **public** in `Lamp`

```
open class Lamp(~~private~~ var isOn: Boolean) {
    …
```

- ✓ Subclasses of `Lamp` can read `isOn`
- ? Subclasses of `Lamp` can modify `isOn`
- ✗ **Any code, anywhere**, can **read and modify** `isOn`

This maximally violates encapsulation

The **protected** visibility modifier

Remember: a property or method of a class can be:

- **public** – visible everywhere in the codebase
- **private** – only visible inside the class

Another visibility modifier, specific to inheritance:

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

One more visibility modifier, **internal**, not covered in this course

Visibility modifiers: summary

Class level:

Applies to:

- properties
- methods
- nested classes
- inner classes
- nested interfaces

Visible in …

	Same class	Subclass	Entire codebase
`private`	✓	✗	✗
`protected`	✓	✓	✗
`public` (default)	✓	✓	✓

Top level:

Applies to:

- top-level variables
- top-level functions
- classes
- interfaces

Visible in …

	Same file	Entire codebase
`private`	✓	✗
`public` (default)	✓	✓

Better solution: make `isOn` **protected** in `Lamp`

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

Remember: **protected** visibility is between **public** and **private**

- ✓ Subclasses of `Lamp` can read `isOn`
- ? Subclasses of `Lamp` can modify `isOn`
- ✓ `isOn` is not visible except in `Lamp` and its subclasses

Much better, but still slightly violates encapsulation

Best solution: **protected** `get` and **private** `set` for `isOn` in `Lamp`

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

✓ Subclasses of `Lamp` can read `isOn`

✓ Subclasses of `Lamp` cannot modify `isOn`

✓ `isOn` is not visible except in `Lamp` and its subclasses

Best: `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 Lamp class, ready to be extended

```
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)"
        }
}
```

The `DimmingLamp` subclass (same as earlier)

```kotlin
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 {
                ""
            }
}
```

Better style: avoids duplicate code and “magic numbers”

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

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

    override fun pressSwitch() {
        super.pressSwitch()
        brightness = initialBrightness()
    }

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

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

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

    private fun initialBrightness() =
        if (isOn) { MAX_BRIGHTNESS } else { 0 }
}
```

Inheritance: terminology

A subclass:

- **Extends** a superclass
- **Derives** from a superclass
- **Specialises** a superclass
- **Inherits** from a superclass

Interchangeable terminology

There should be an “is a” relationship between subclass and superclass
- a subclass instance “is a” superclass instance

A `DimmingLamp` **is a** `Lamp`

Inheritance: terminology

A superclass:

- **Generalises** a subclass

A superclass may also be called:

- Parent class
- Base class

A subclass may also be called:

- Child class
- Derived class

Inheritance: terminology

Inheritance is **transitive**:

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

We sometimes say:

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

Inheritance: terminology

Properties and methods of a class are often referred to collectively as **members** of the class

A subclass **inherits** the public and protected members of its superclasses

GridWorld game

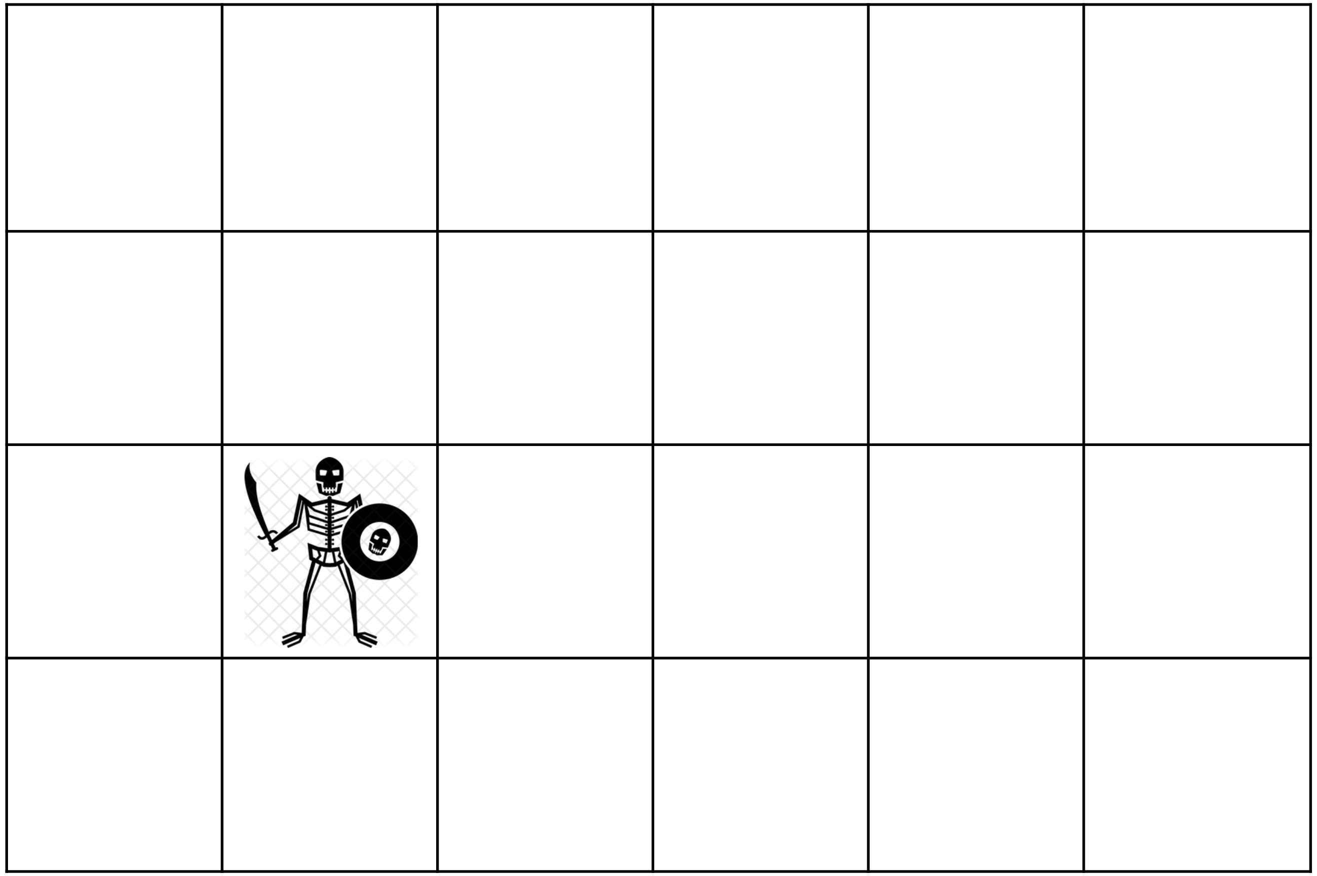
Player

GridWorld game

Player

Navigates grid
of squares

Various kinds of terrain:



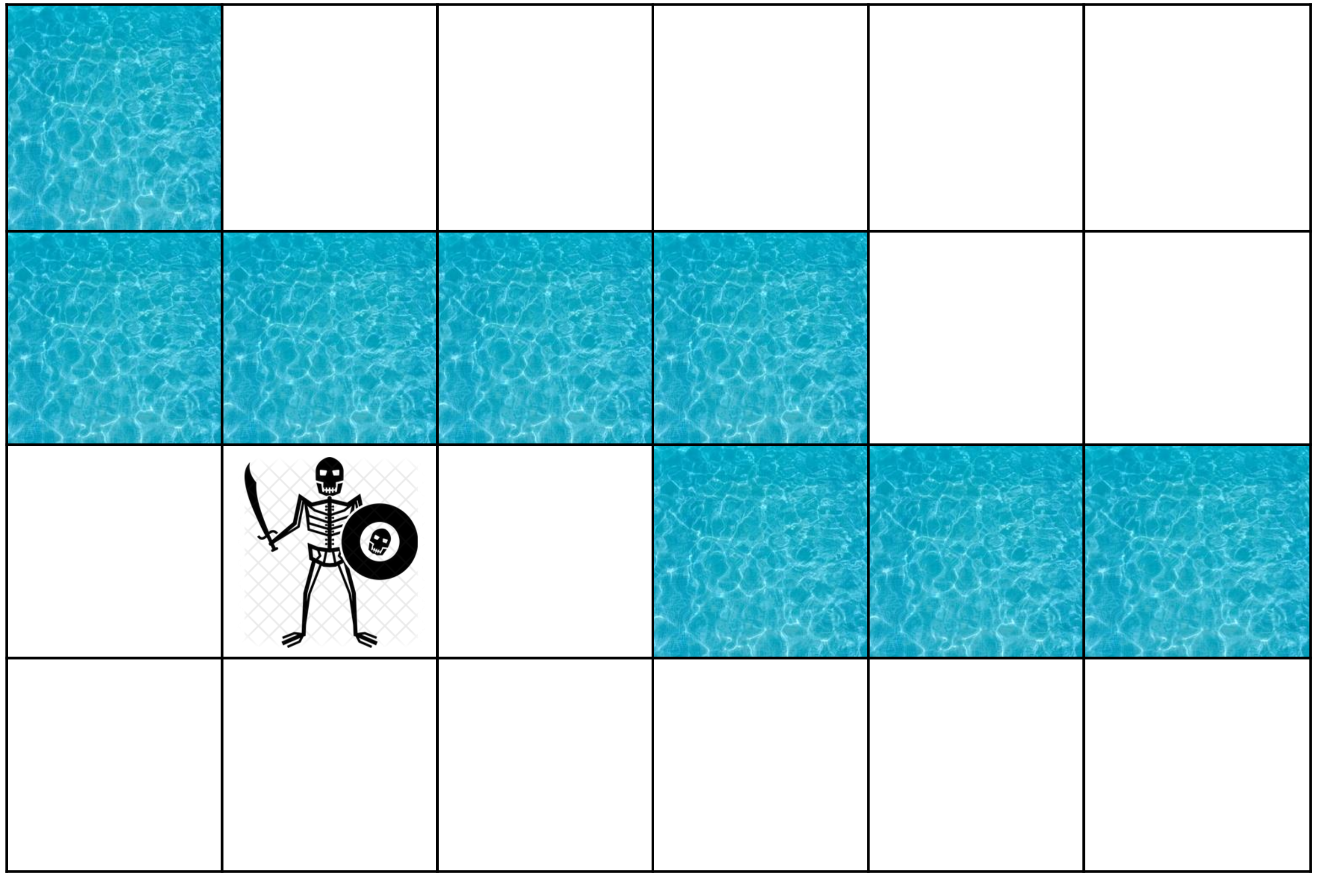
`GridWorld` game

Player

Navigates grid of squares

Various kinds of terrain:

- Water



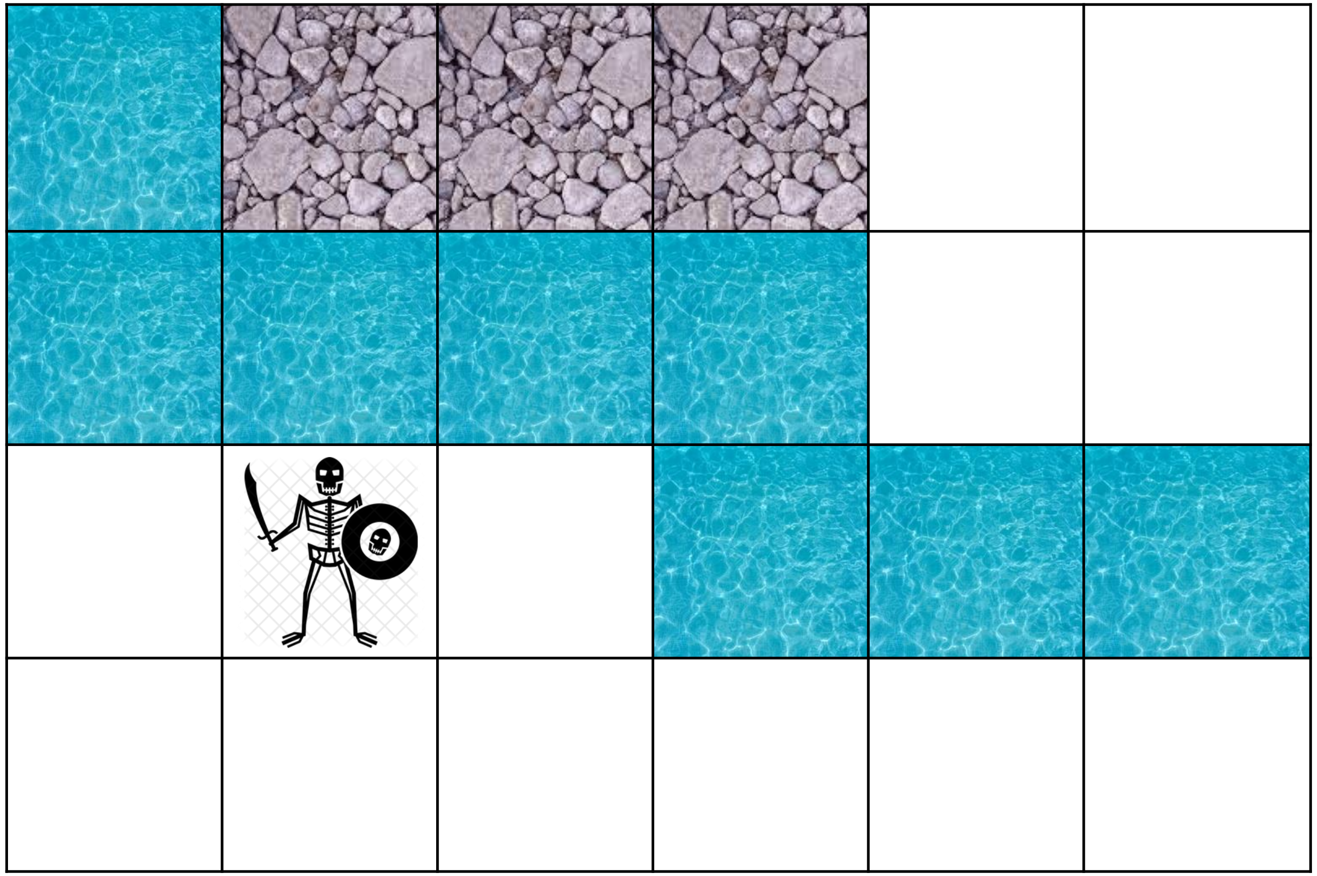
`GridWorld` game

Player

Navigates grid of squares

Various kinds of terrain:

- Water
- Rocks



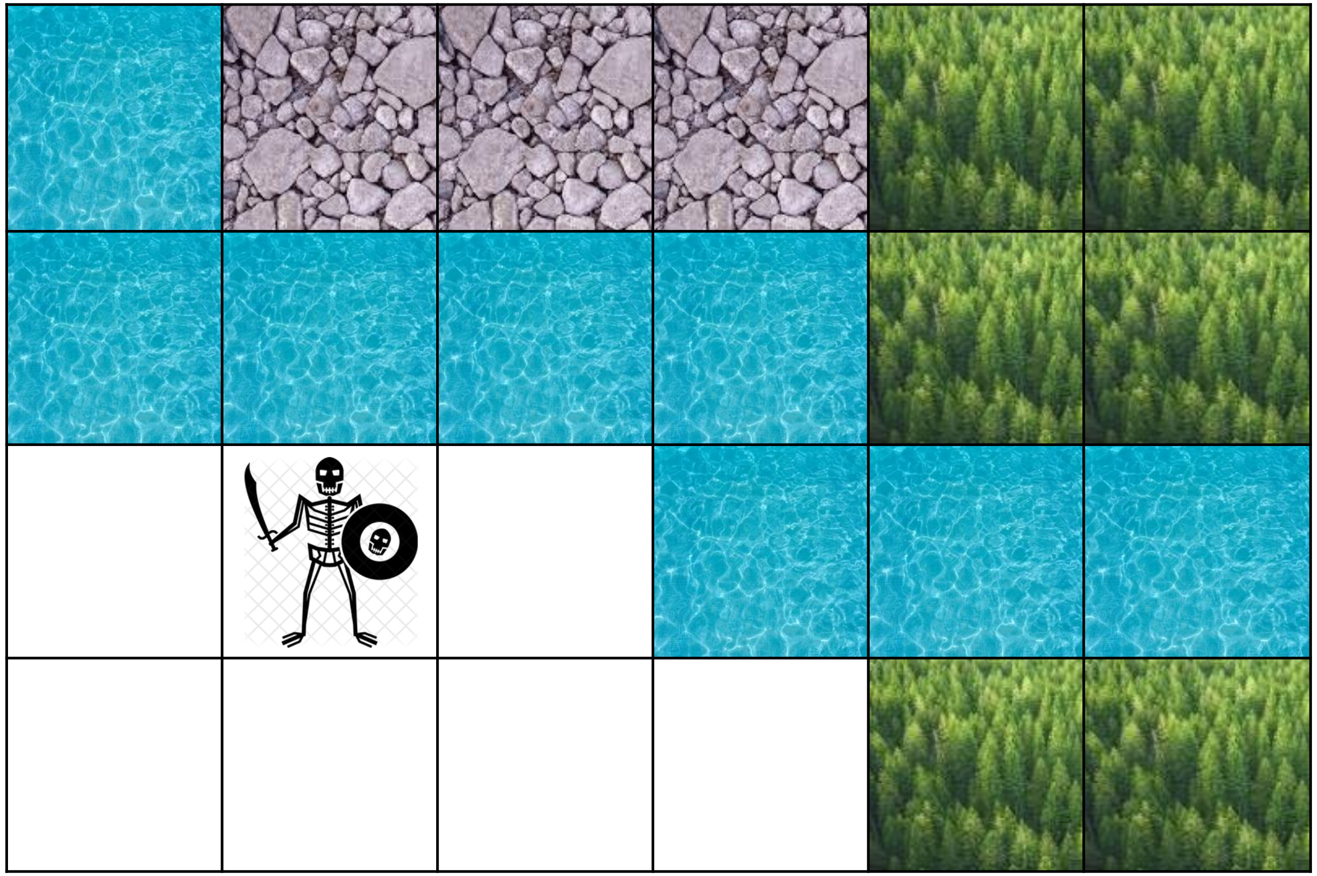
`GridWorld` game

Player

Navigates grid of squares

Various kinds of terrain:

- Water
- Rocks
- Forest



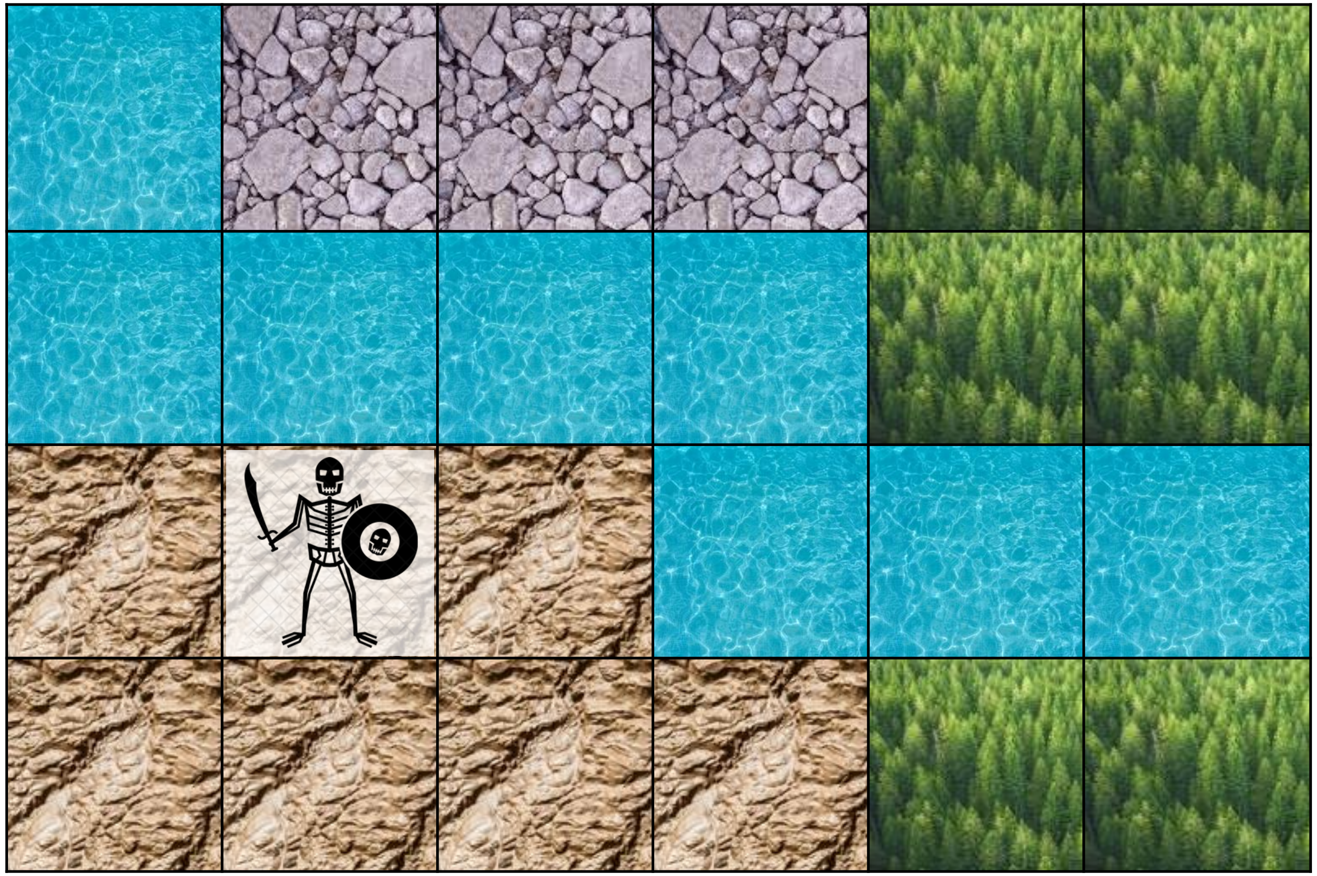
`GridWorld` game

Player

Navigates grid of squares

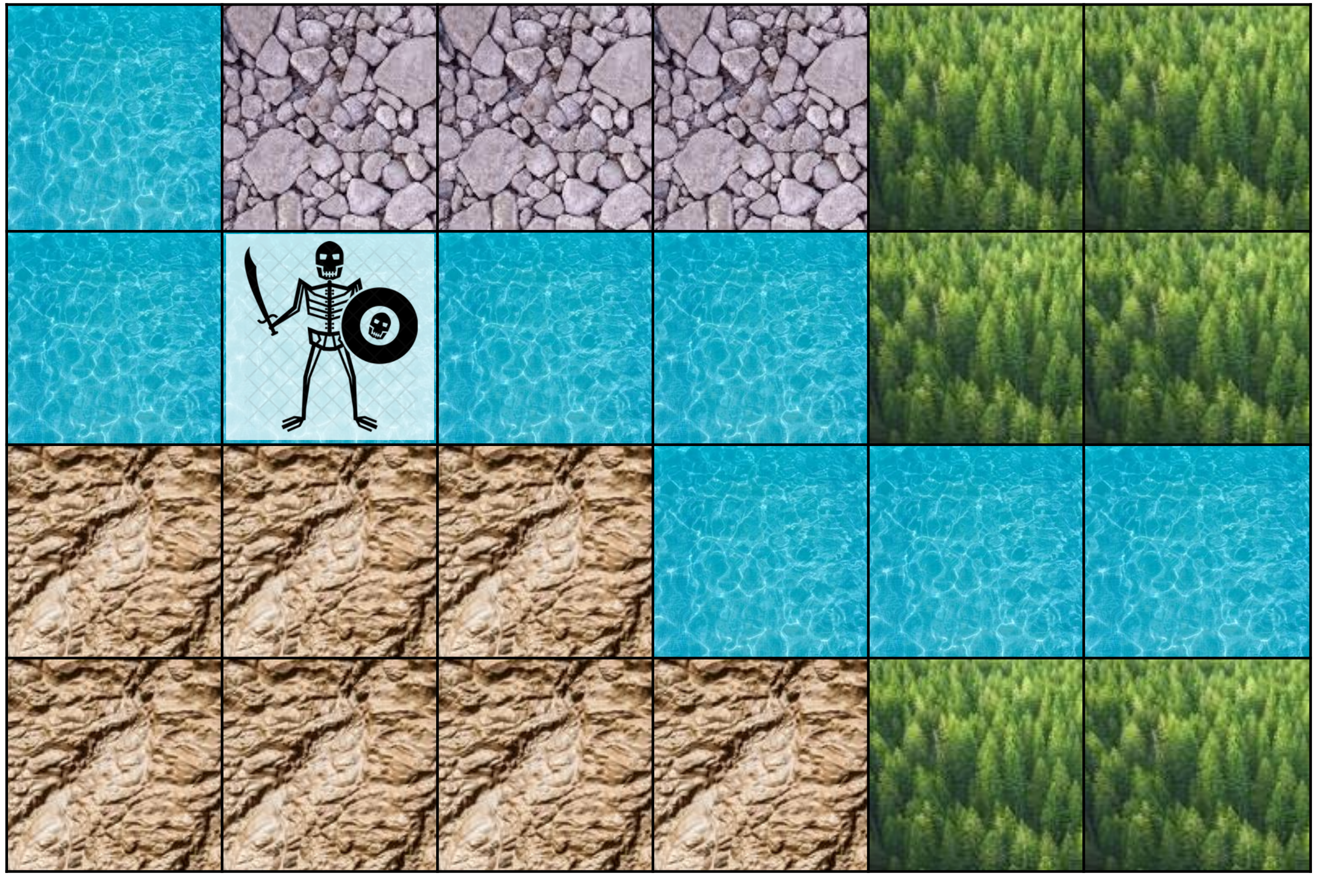
Various kinds of terrain:

- Water
- Rocks
- Forest
- Swamp



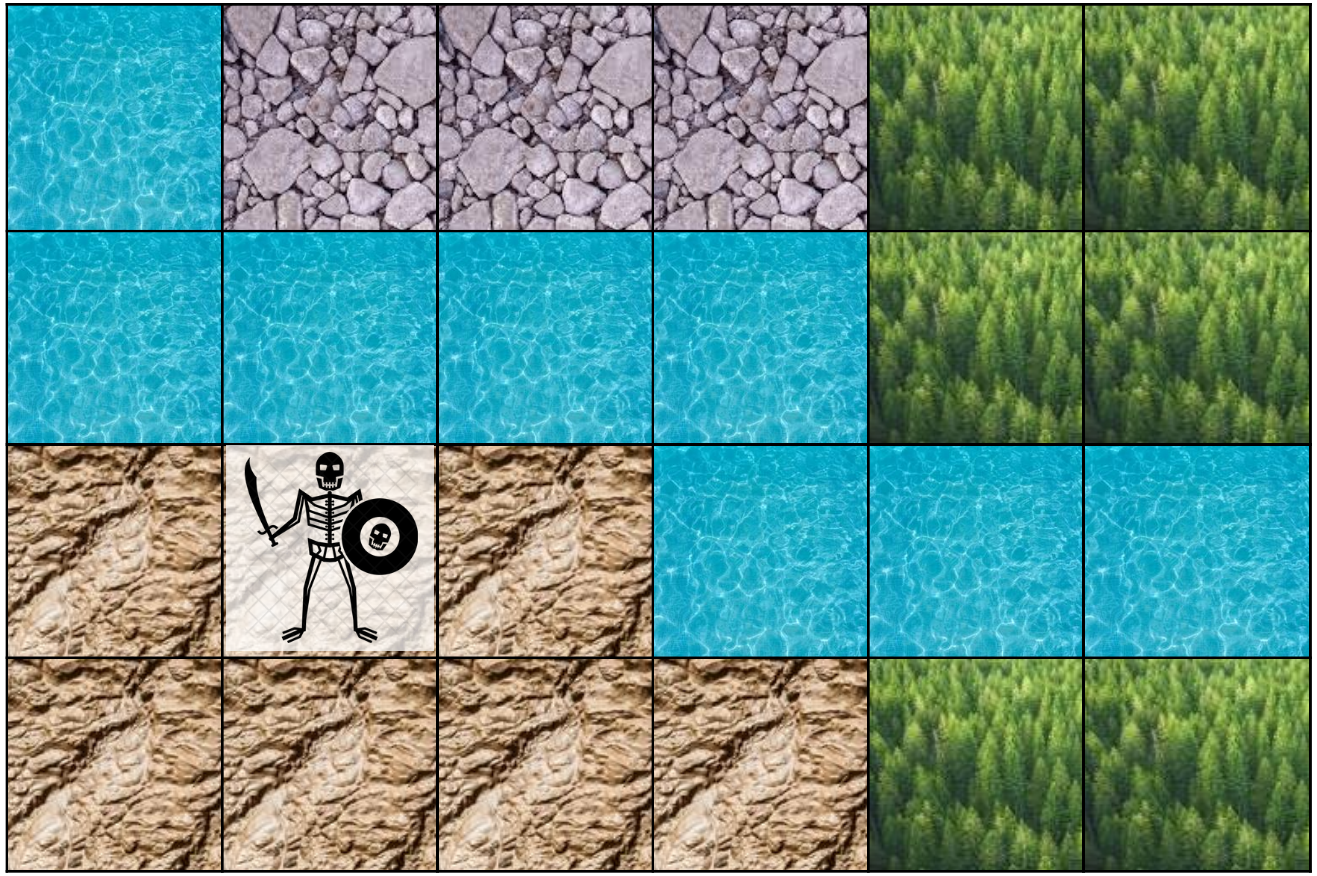
GridWorld game

Player can move **up**



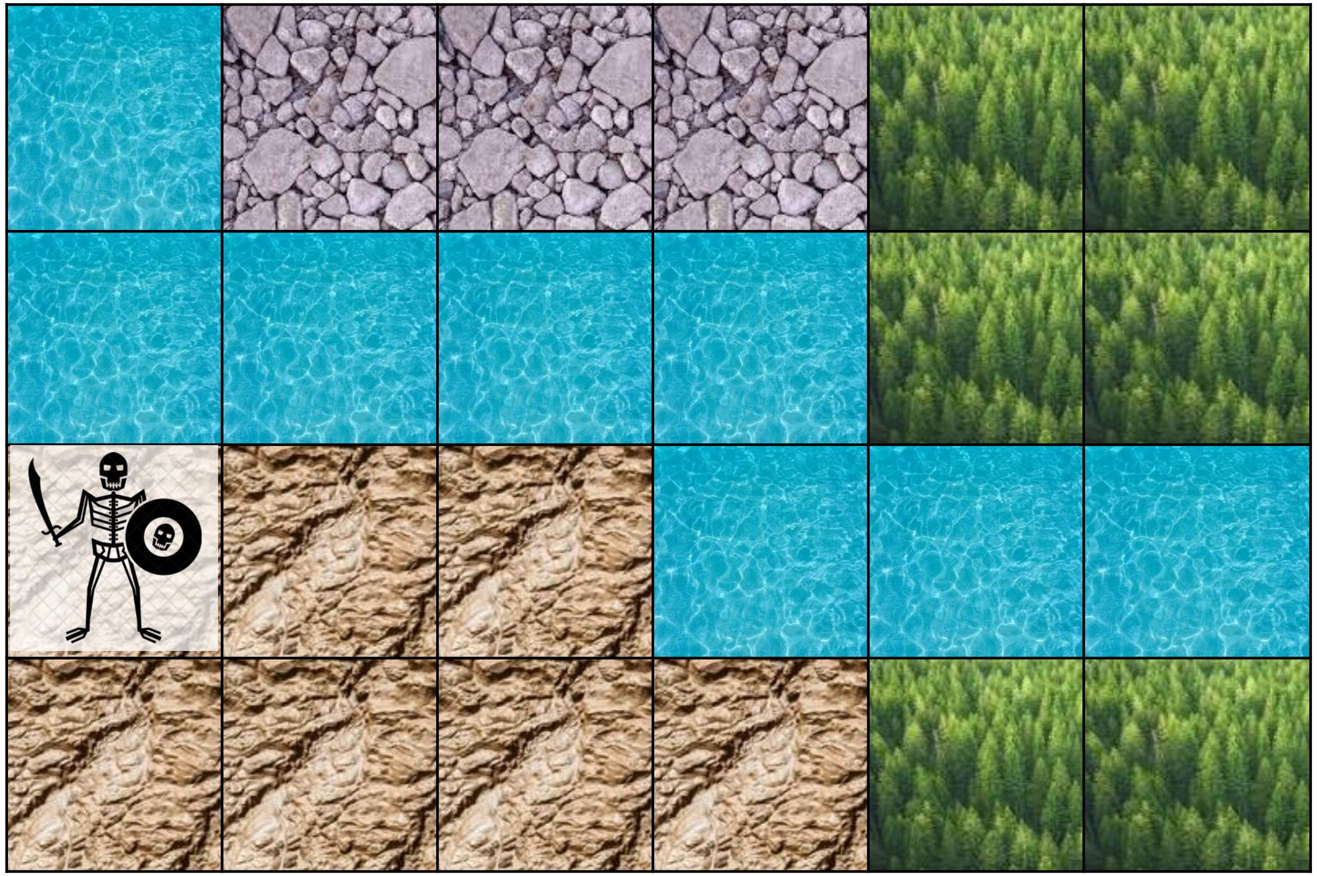
`GridWorld` game

Player can move **down**



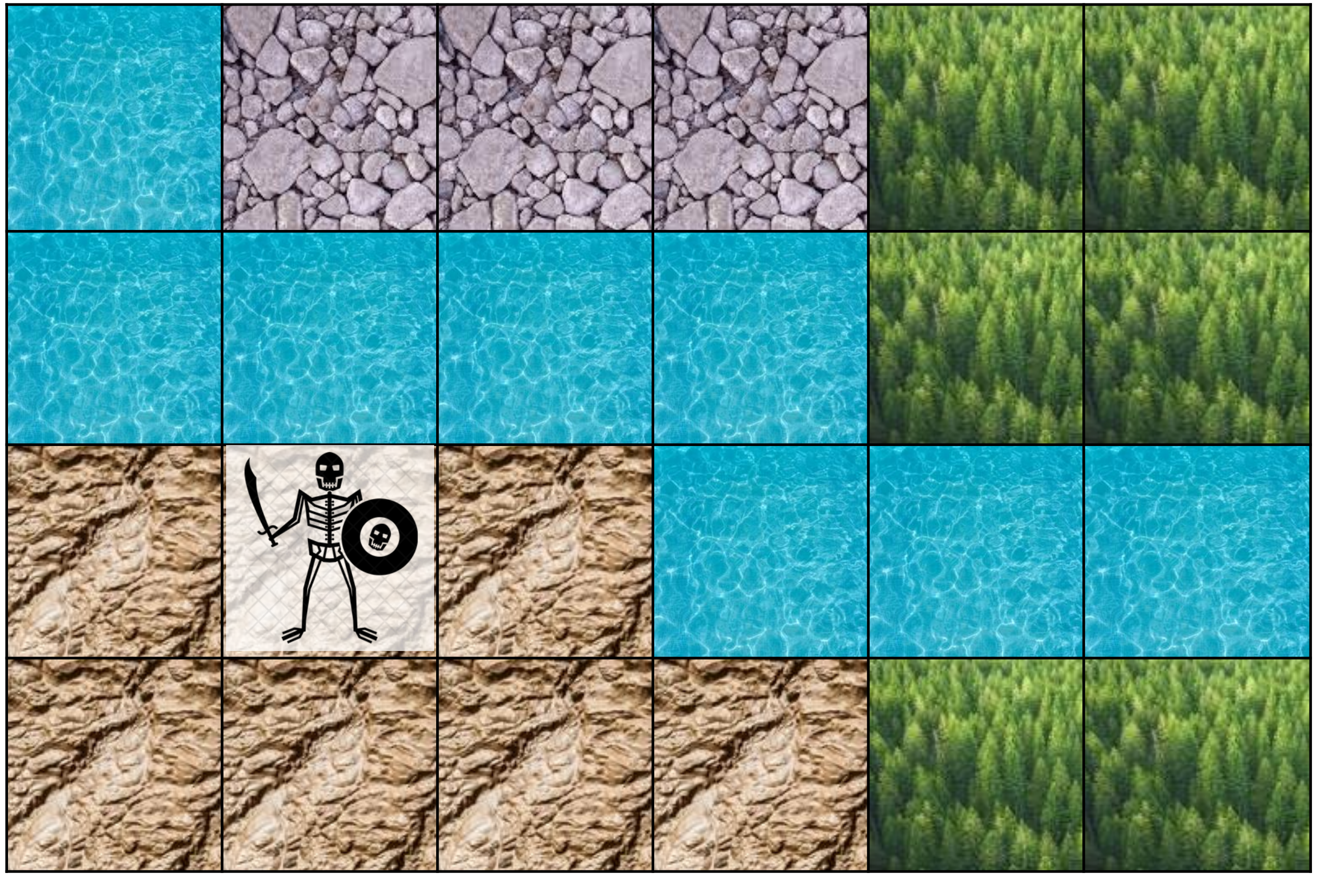
`GridWorld` game

Player can move **left**



GridWorld game

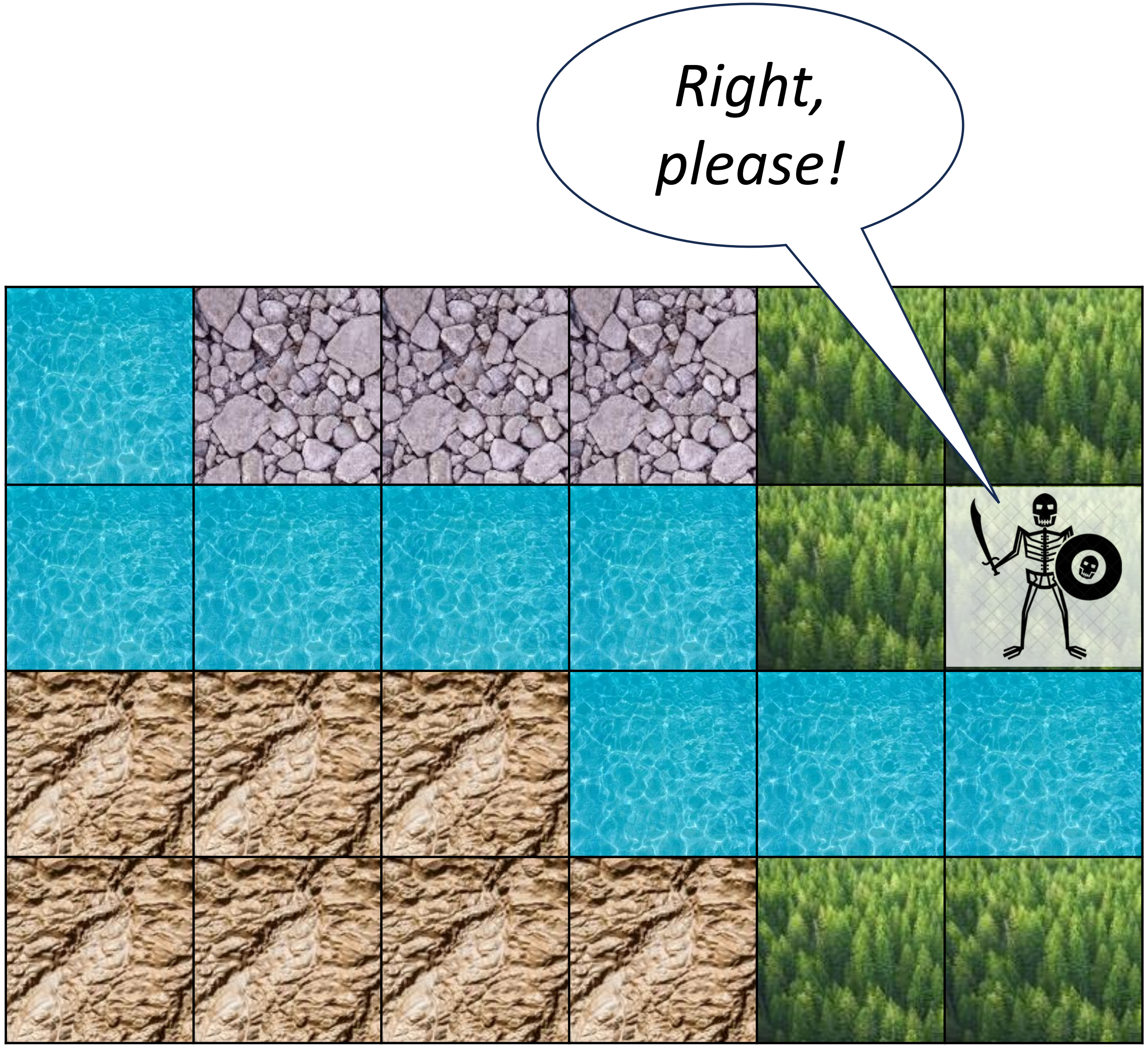
Player can move **right**



`GridWorld` game

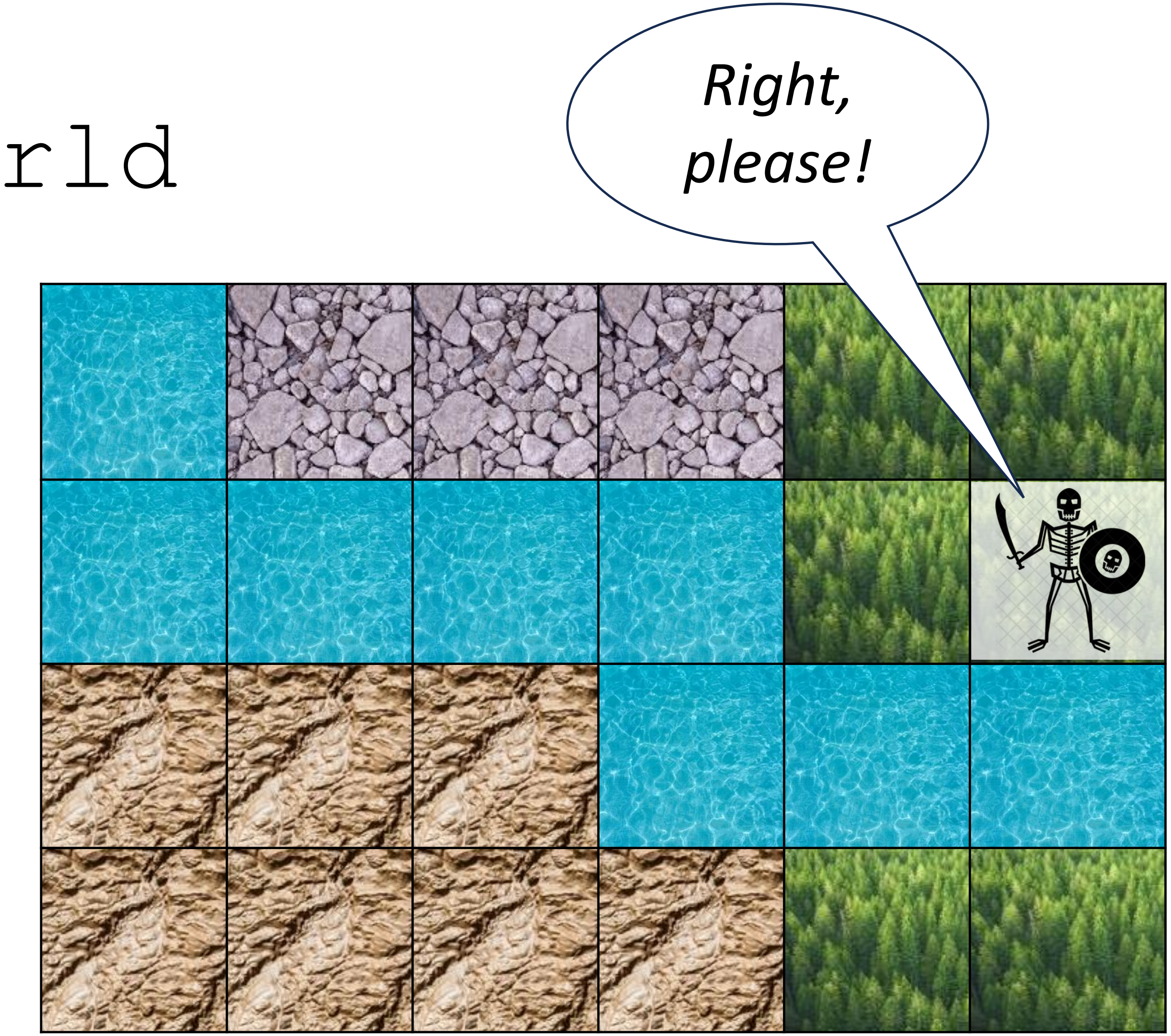
What happens when player tries to move off edge of grid?

Three kinds of world…



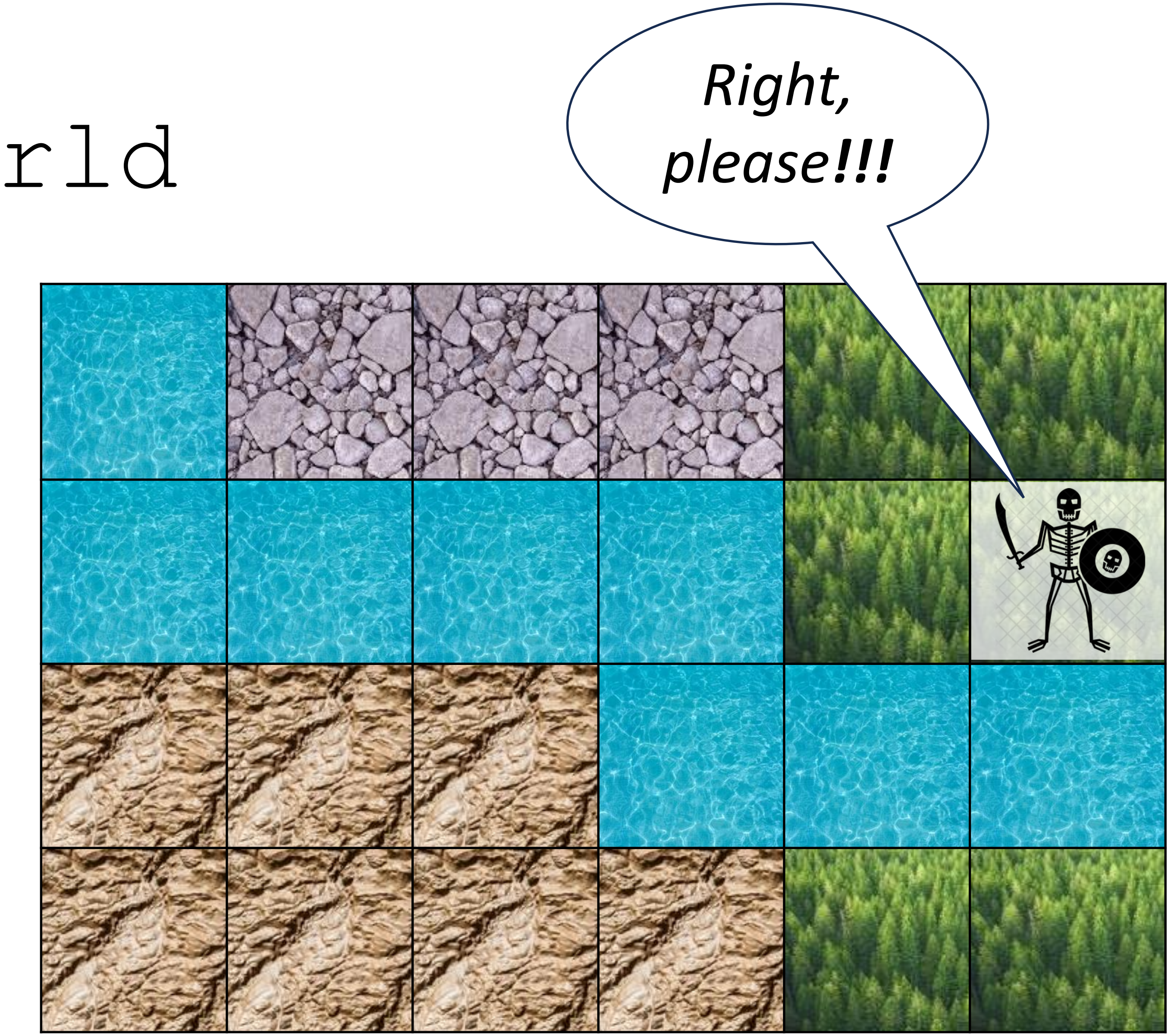
Bounded `GridWorld`

Attempting to move off the grid has **no effect**



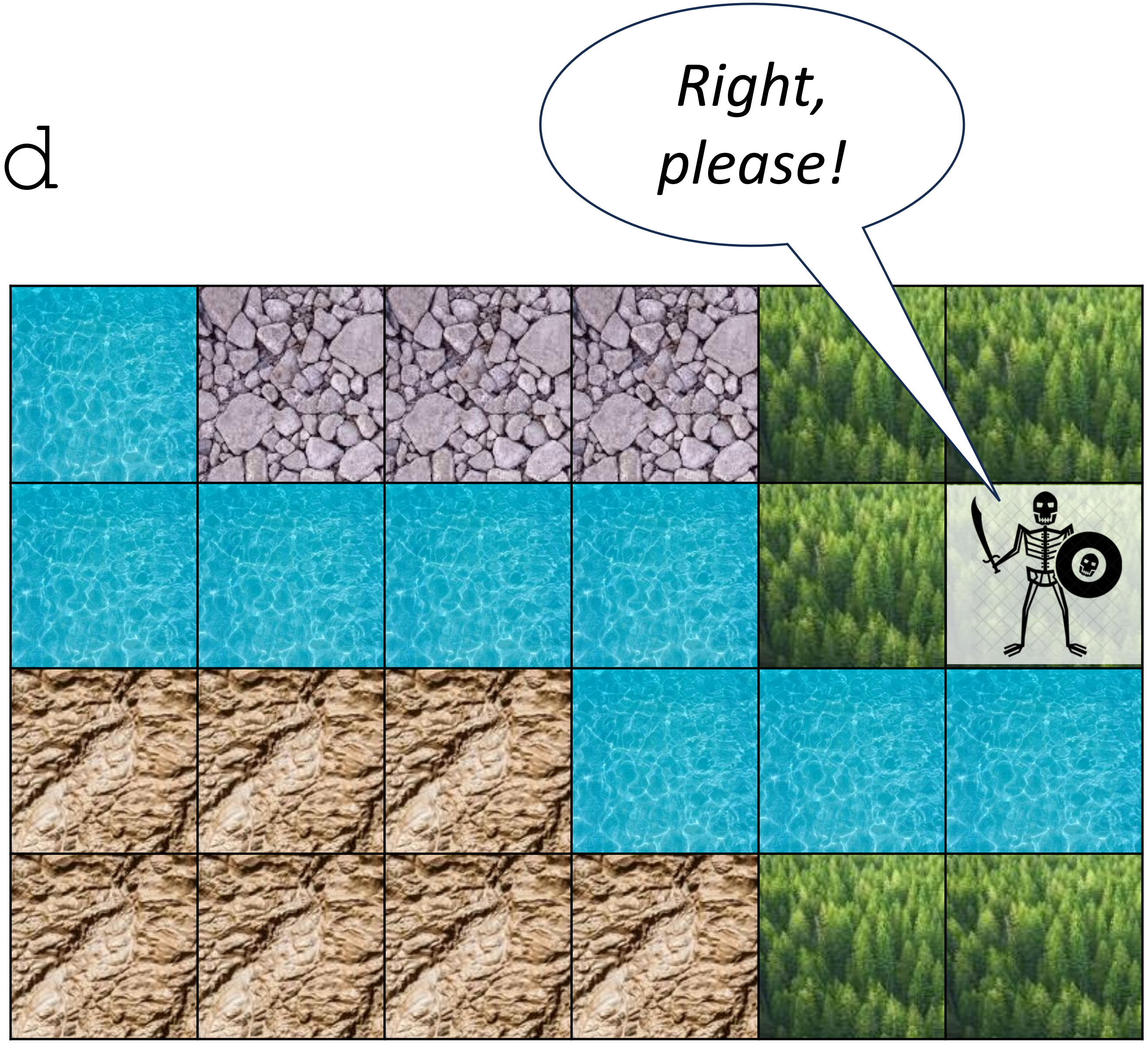
Bounded `GridWorld`

Attempting to move off the grid has **no effect**



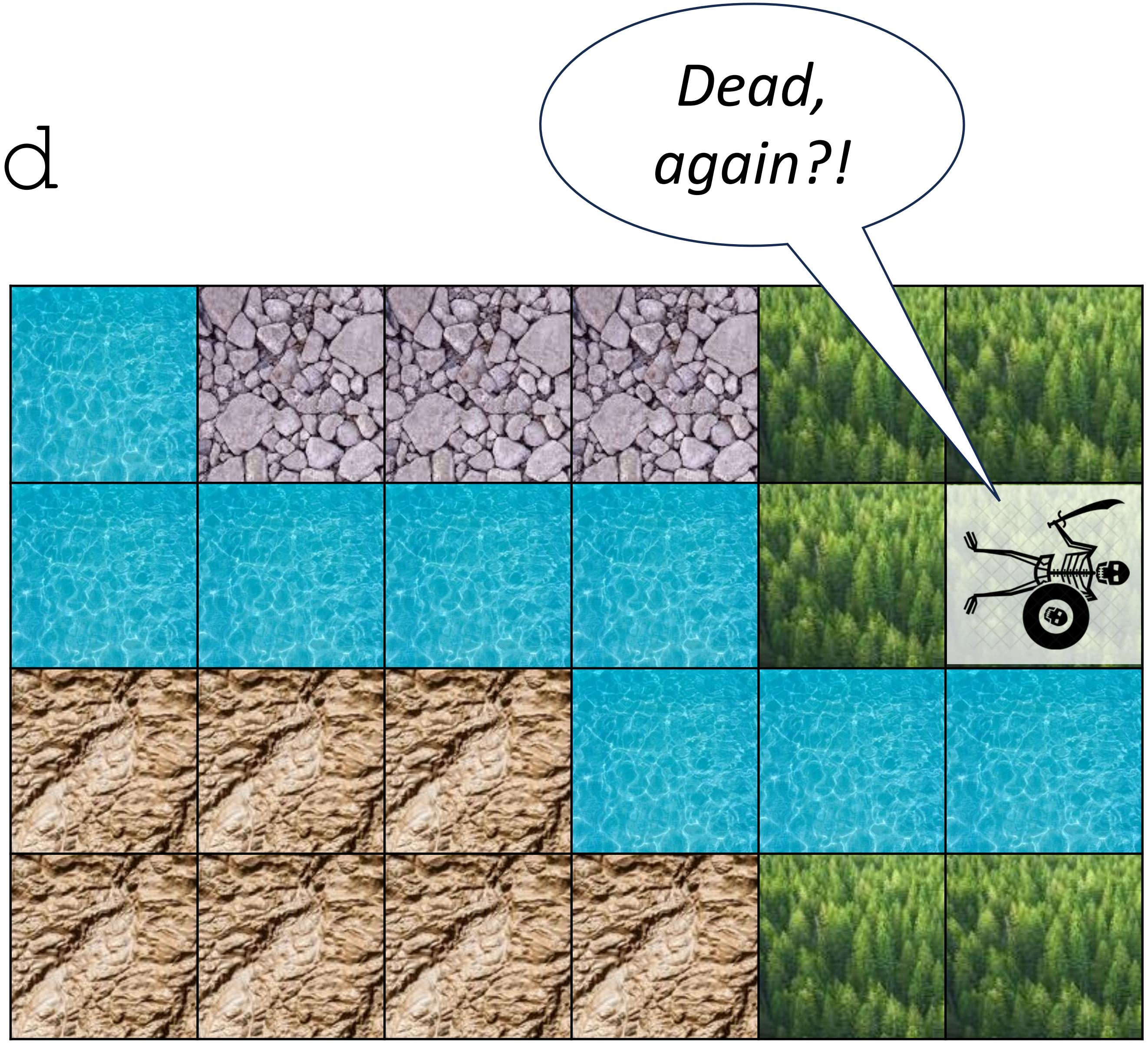
Deadly `GridWorld`

Attempting to move off the grid leads to **sudden death**



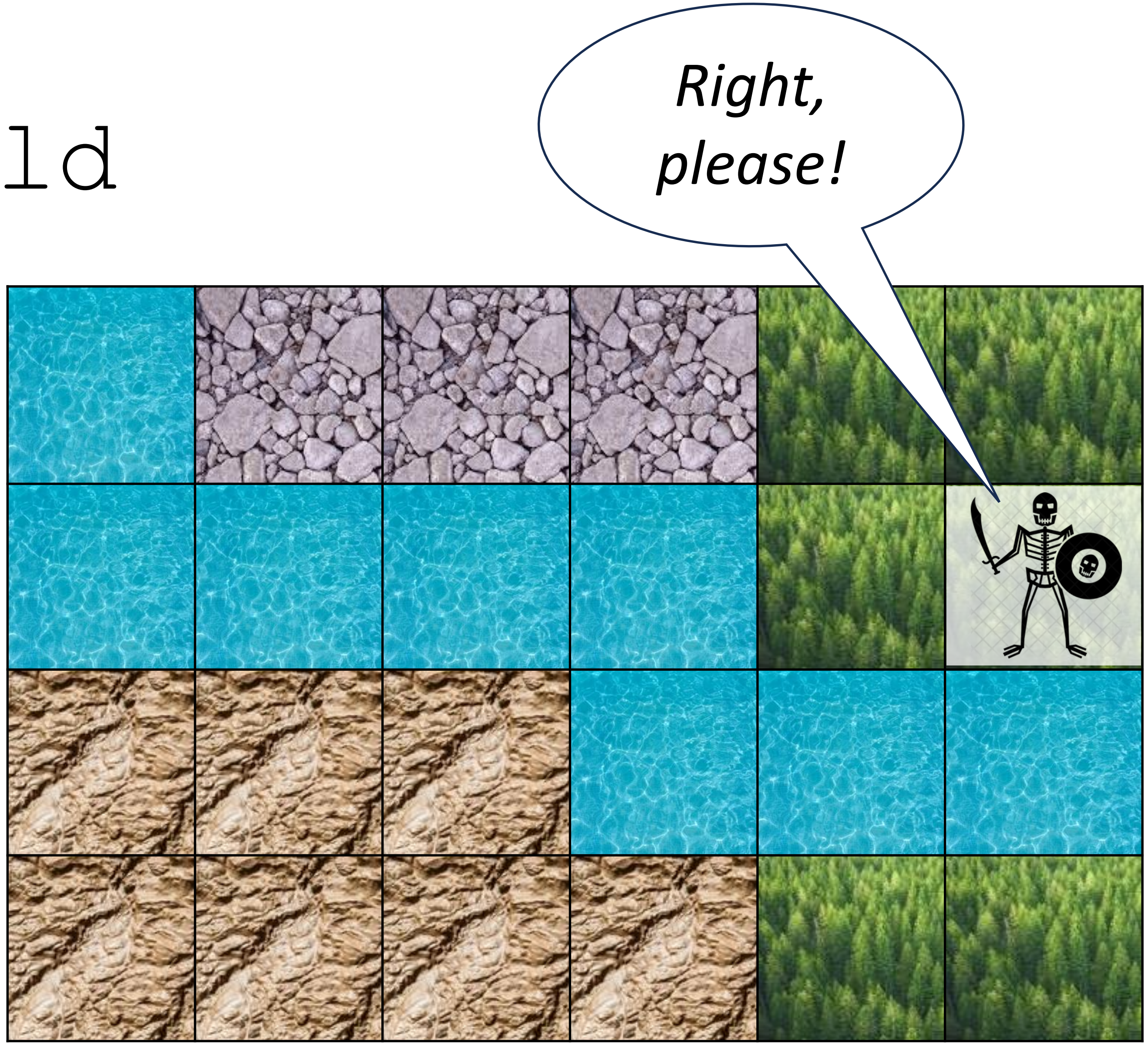
Deadly `GridWorld`

Attempting to move off the grid leads to **sudden death**



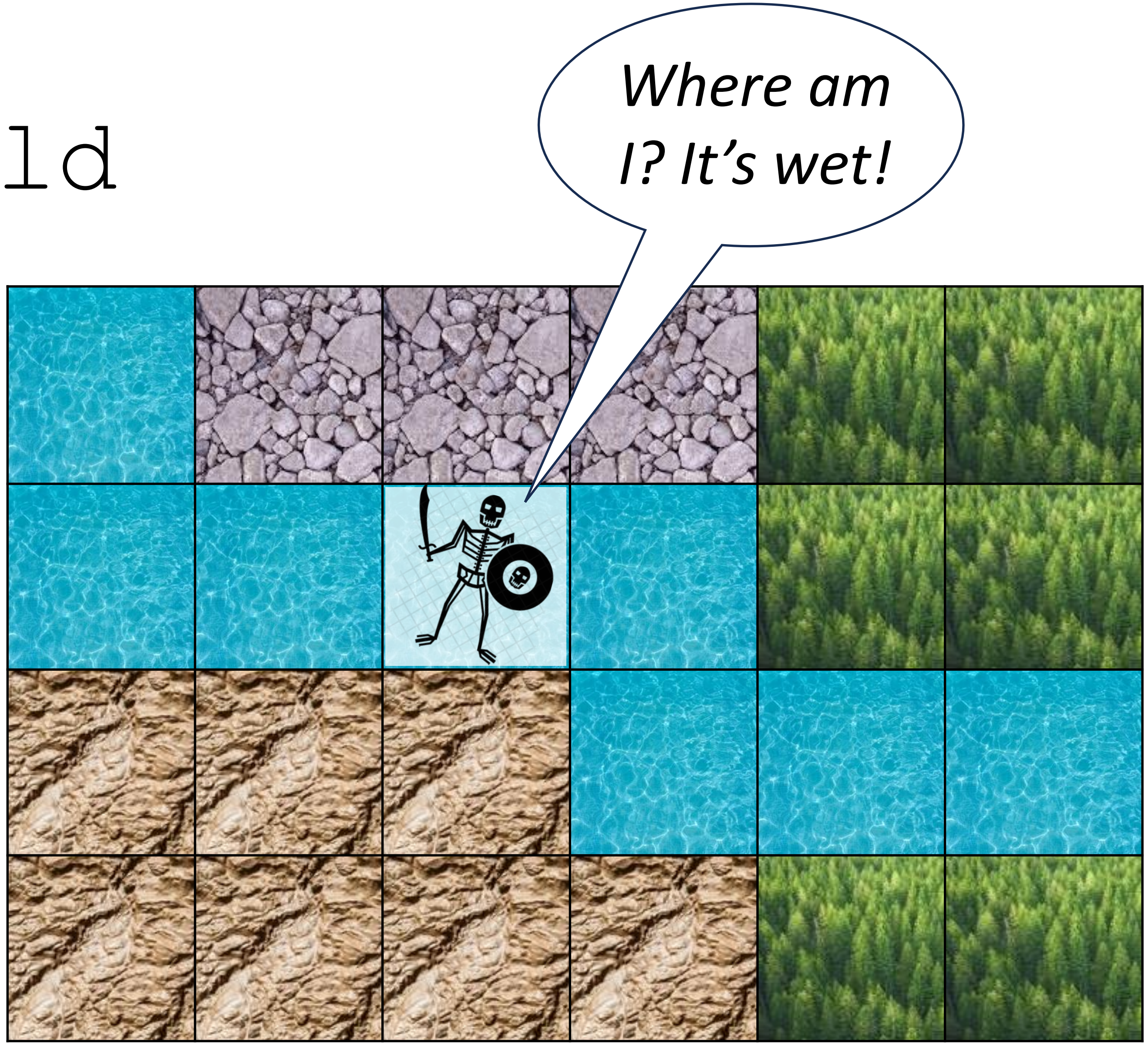
Random `GridWorld`

When player moves off the grid the **teleport** to a random square



Random `GridWorld`

When player moves off the grid the **teleport** to a random square



First attempt at building `GameWorld`

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

enum class WorldKind {
    BOUNDED, DEADLY, RANDOM
}

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

Inheritance: To define your own exceptions you subclass the `Exception` class

```
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 up with a way of showing the game world as text

Problem with this design

Not extensible: the world kinds need to be known upfront

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

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

```
class GridWorld(…) {
    …
    private fun updatePosition(newPosition: Pair<Int, Int>) {
        …
        when (worldKind) {
            WorldKind.BOUNDED -> position = clampToGrid(newPosition)
            WorldKind.DEADLY -> throw DeadPlayerException("Fell of world!")
            WorldKind.RANDOM -> position = randomPosition()
        }
    }
    …
}
```

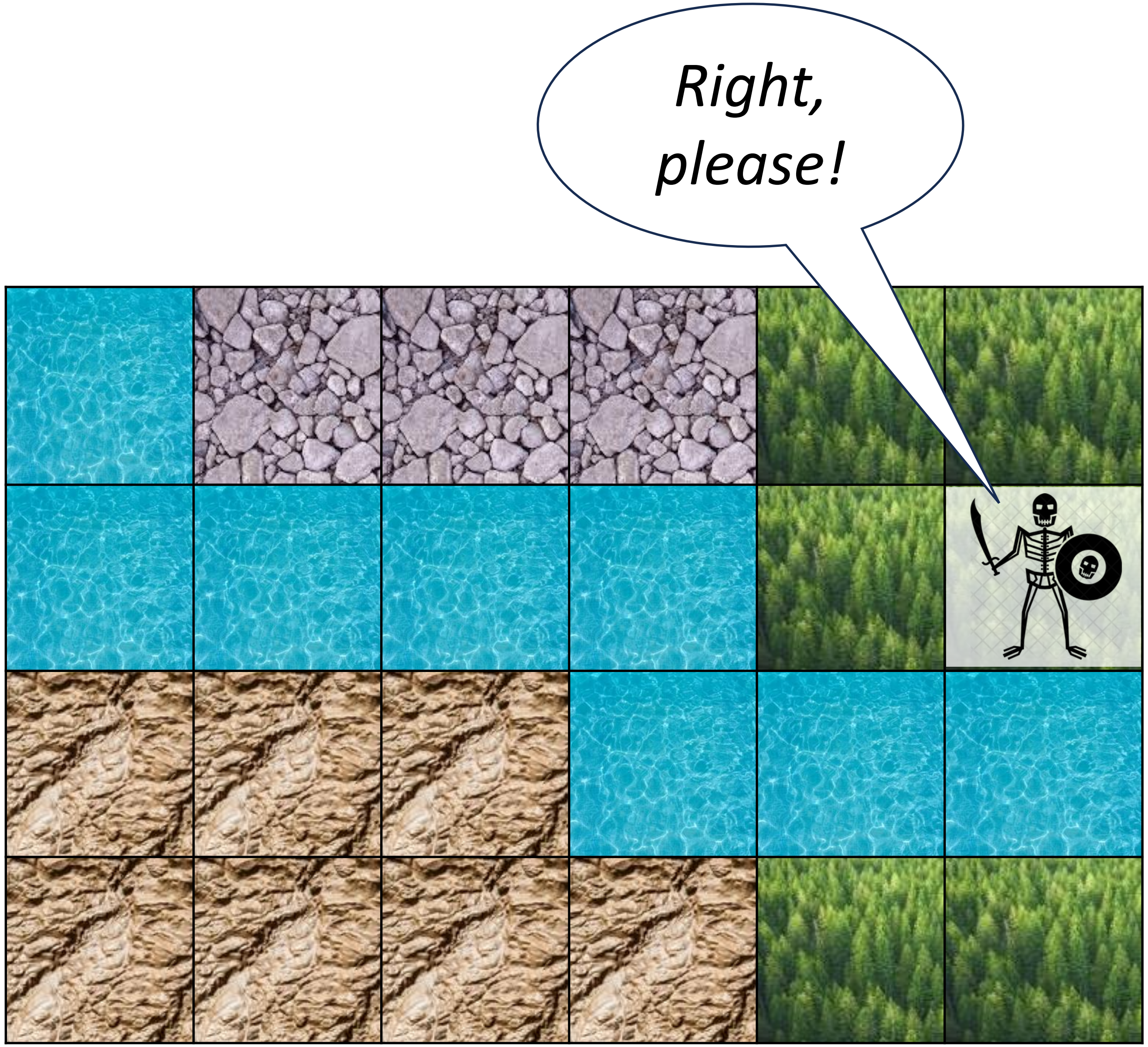
Special code for each kind of world

Problem with this design

What if we add a new kind of world?

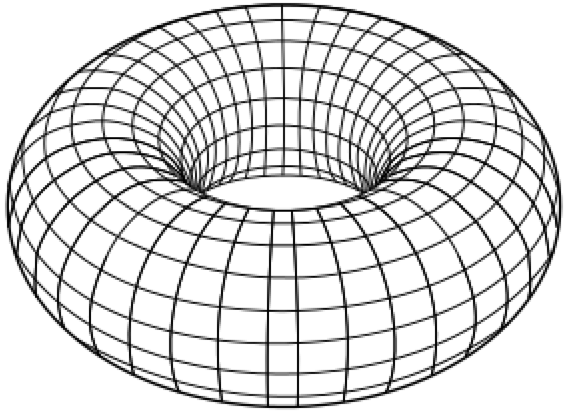
```
enum class WorldKind {
    BOUNDED, DEADLY, RANDOM, TORUS
}
```

Torus `GridWorld`



The grid has wrap-around behaviour:

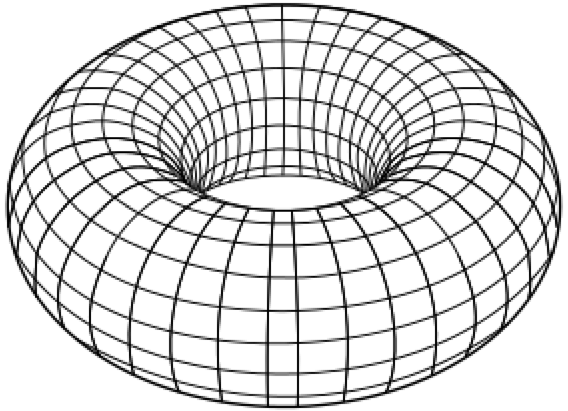
- moving off the right takes player to the left
- moving off the top takes player to the bottom
- etc.

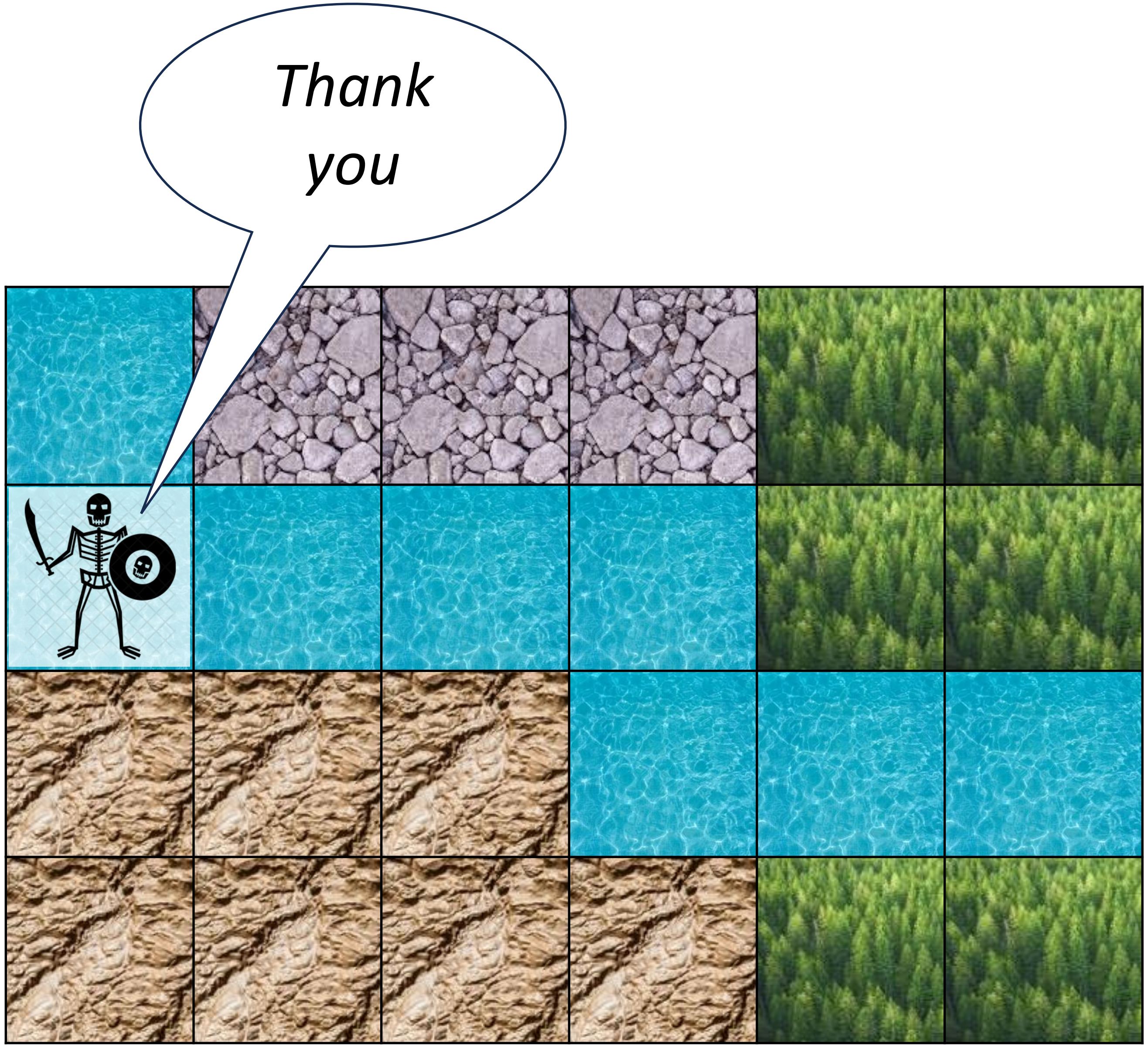


Torus `GridWorld`

The grid has wrap-around behaviour:

- moving off the right takes player to the left
- moving off the top takes player to the bottom
- etc.





Problem with this design

What if we add a new kind of world?

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

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

```
class GridWorld(…) {
    …
    private fun updatePosition(newPosition: Pair<Int, Int>) {
        …
        when (worldKind) {
            WorldKind.BOUNDED -> position = clampToGrid(newPosition)
            WorldKind.DEADLY -> throw DeadPlayerException("Fell of world!")
            WorldKind.RANDOM -> position = randomPosition()
        }
    }
    …
}
```

Compile error: 'when' expression must be exhaustive, add necessary 'TORUS' branch or 'else' branch instead

This error is **useful**: forces us to update the game. But: it would be better if the game was more naturally extensible.

Alternative design – **inheritance**

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

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

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

Let’s get rid of the `WorldKind` enumeration

Allows subclasses

Allows properties to be accessed by subclasses

```
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")
    …
}
```

No more `worldKind` parameter

Can be **overridden** by subclasses of `GridWorld`

Only visible to `GridWorld` and subclasses

Subclassess for different kinds of worlds will define what happens when there is an overrun

The `GridWorld` superclass does not know how to handle an overrun

Throwing an error is a **hack** – we will see a better approach soon!

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)),
        )
}
```

These refer to the `width` and `height` properties of `GridWorld`, which are **inherited**. Because they are protected, they are visible to `BoundedGridWorld`.

DeadlyGridWorld subclass

```kotlin
class DeadlyGridWorld(
    width: Int,
    height: Int,
) : GridWorld(width, height) {
    override fun handleOverrun(newPosition: Pair<Int, Int>): Pair<Int, Int> =
        throw DeadPlayerException("Fell off world!")
}
```

`DeadlyGridWorld` subclass

Alternative: 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!")
}
```

This override of `handleOverrun` does not return anything – it unconditionally throws an exception

We can document this by narrowing the return type to `Nothing` – the Kotlin type with **no values**

Exercise

Write `RandomGridWorld` and `TorusGridWorld` subclasses

Inheritance-based design: problem 1

We do not want a “plain” `GridWorld` object, but nothing stops a client creating one

This is not any particular kind of `GridWorld`

```
fun main() {
    val strangeWorld = GridWorld(10, 10)
    for (i in 1..10) {
        strangeWorld.left()
    }
}
```

It would be better if we could not create “just a `GridWorld`”

Output:

```
kotlin.NotImplementedError: This method
should be provided by subclasses
```

Inheritance-based design: problem 2

Nothing forces us to override the dummy superclass method

```
class TorusGridWorld(
    width: Int,
    height: Int,
) : GridWorld(width, height) {
// TODO - come back to this once I read about what a torus is
//    override fun handleOverrun(newPosition: Pair<Int, Int>): Pair<Int, Int> =
//        ...
}

fun main() {
    val doughnutWorld = TorusGridWorld(10, 10)
    for (i in 1..10) {
        doughnutWorld.left()
    }
}
```

We forgot to come back and finish this off

It would be nice if the compiler **forced us** to implement this method

Output:

```
kotlin.NotImplementedError: This method
should be provided by subclasses
```

Inheritance-based design: problem 3 (minor)

The dummy superclass `handleOverrun` implementation is available via `super`

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

Accidental superclass call – leads to exception

It would be better if this call was not allowed

Solution: make `GridWorld` an **abstract** class

Solution: make `GridWorld` an **abstract** class

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











}
```

`abstract` before `class` means: “this is an abstract class – you cannot create direct instances of this class”

An abstract class is automatically **open**: the entire point of an abstract class is to support subclasses – a **final** abstract class would serve no purpose

Solution: make `GridWorld` an **abstract** class

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

`abstract` before `fun` means: “this is an abstract method – it has no default implementation, and concrete subclasses **must** provide an implementation”

An abstract method is automatically **open**

Same as for abstract methods of interfaces

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

Solution: make `GridWorld` an **abstract** class

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

Abstract classes can have concrete properties and methods

Solution: make `GridWorld` an **abstract** class

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

A concrete method of an abstract class can be defined in terms of abstract methods

Problem 1: solved

We **cannot** create a “plain” `GridWorld` object

```
fun main() {
    val strangeWorld = GridWorld(10, 10)
    …
}
```

Compiler error: Cannot create an instance of an abstract class

Excellent!

Problem 2: solved

We **must** implement the abstract method

```
class TorusGridWorld(
    width: Int,
    height: Int,
) : GridWorld(width, height) {
// TODO - come back to this once I read about what a torus is
//    override fun handleOverrun(newPosition: Pair<Int, Int>): Pair<Int, Int> =
//        ...
}
```

Compile error: Class 'TorusGridWorld' is not abstract and does not implement abstract base class member protected abstract fun handleOverrun(newPosition: Pair<Int, Int>): Pair<Int, Int> defined in demos.GridWorld

Excellent!!

Problem 3 (minor): solved

There is no dummy handleOverrun implementation in superclass

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

Accidental superclass call – leads to **compile error**

Compile error: Abstract member cannot be accessed directly

Excellent!!!

To be continued in part 2

But first: **concurrency**