

# Method Types and Properties

---

Dirk Riehle, Univ. Erlangen

**ADAP B01**

Licensed under [CC BY 4.0 International](https://creativecommons.org/licenses/by/4.0/)

# Agenda

---

## 1. Method types

- a. Query methods
- b. Mutation methods
- c. Helper methods

## 2. Method properties

- a. Implementation properties
- b. Inheritance properties
- c. Convenience methods

## 3. Design guidelines

## Homework

# Professional Language

---

To become a proficient developer, you need to learn the language

- Not just a programming language, but how developers talk about code
- Some of it can be found in textbooks, some cannot
- It is always evolving so stay current

# 1. Method Types

---

# Method Types

---

A method type classifies a method into a particular type

- The method type is indicative of the main purpose
- A method may have only one type, not many

(Also: A method should have one purpose)

# Main Categories of Method Types

---

Query methods are

- Methods that return information about the object but don't change its state

Mutation methods are

- Methods that change the object's state but don't provide information back

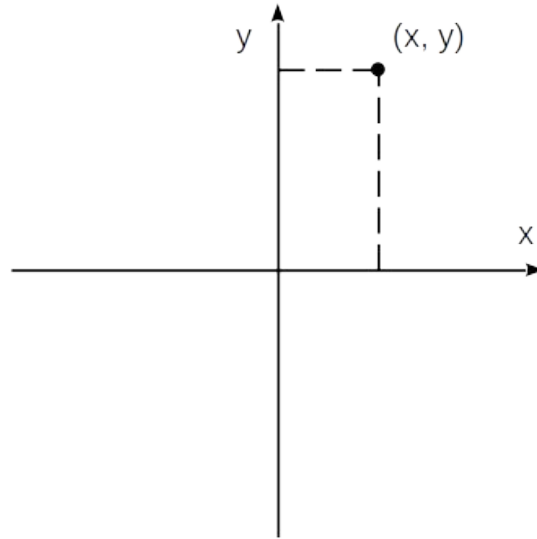
Helper methods are

- Methods that perform some utility function independent of the object

# Classification of Method Types

Query methods	Mutation methods	Helper methods
Get method (getter)	Set method (setter)	Factory method
Boolean query method	Command method	Cloning method
Comparison method	Initialization method	Assertion method
Conversion method	Finalization method	Logging method
...	...	...

# Cartesian Coordinates



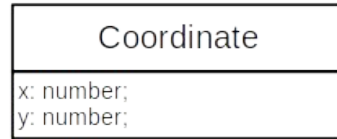
DR



# A Simple Class for Cartesian Coordinates

```
export class Coordinate {  
  
    private x: number = 0;  
    private y: number = 0;  
  
    constructor(x?: number, y?: number) { ... }  
  
    public isEqual(other: Coordinate): boolean { ... }  
  
    public getX(): number { ... }  
    public setX(x: number): void { ... }  
    public getY(): number { ... }  
    public setY(y: number): void { ... }  
  
    public calcStraightLineDistance(other: Coordinate): number { ... }  
  
    ...  
}
```

# Class Model of Cartesian Coordinate



## 2. Query Methods

---

# Get Method (A.k.a. Getter)

---

A get method is

- A query method that returns a logical field of the queried object

Example

- `getX(): number`

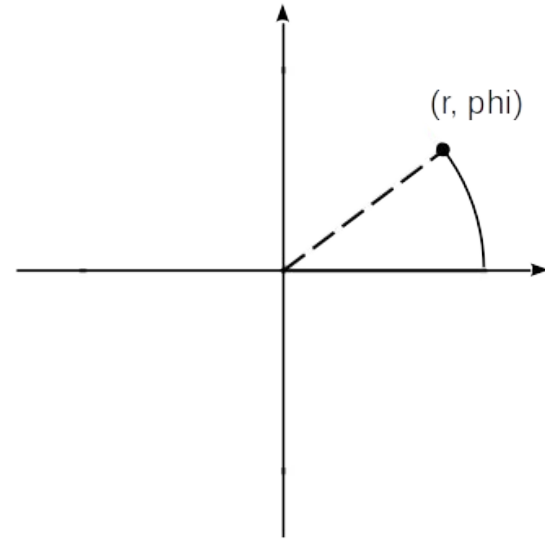
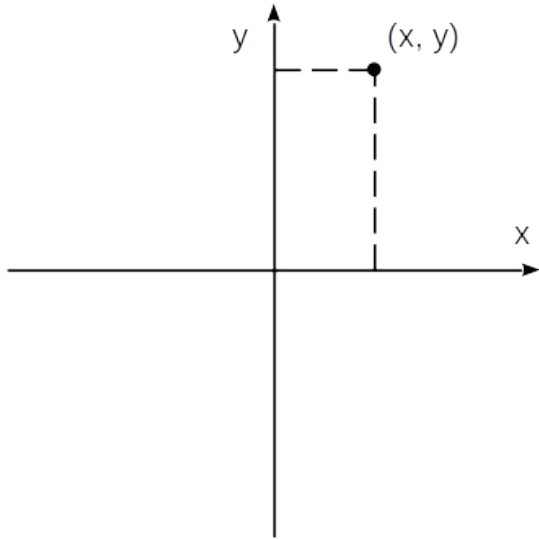
Prefixes

- `get`

Naming

- Prefix + the name of the logical field

# Cartesian and Polar Coordinates



DR

# Logical vs. Physical State

---

Logical state (fields / attributes) is visible in the interface

Physical (“implementation”) state is the actual fields in memory

# Get Method Examples

```
public getX(): number {  
    return this.x;  
}
```

```
public getY(): number {  
    return this.y;  
}
```

```
public getR(): number {  
    return Math.hypot(this.getX(), this.getY());  
}
```

```
public getPhi(): number {  
    return Math.atan2(this.getY(), this.getX());  
}
```

# Boolean Query Method

---

A boolean query method is

- A query method that returns boolean state about the queried object

Example

- `isEqual()`: boolean

Prefixes

- `is`

Naming

- Prefix + the name of the state



# Boolean Query Method Examples

---

```
public isEqual(other: Coordinate): boolean {  
    return (this.getX() == other.getX()) && (this.getY() == other.getY());  
}
```

# Equality

---

```
export interface Equality {  
  isEqual(other: any): boolean;  
  getHashCode(): number;  
}
```

# Equality Contract

---

Two equal objects must have the same hashcode

# Equality Implementation

```
public isEqual(other: Coordinate): boolean {  
    return (this.getX() == other.getX()) && (this.getY() == other.getY());  
}
```

```
public getHashCode(): number {  
    let hashCode: number = 0;  
    const s: string = this.asDataString();  
    for (let i = 0; i < s.length; i++) {  
        let c = s.charCodeAt(i);  
        hashCode = (hashCode << 5) - hashCode + c;  
        hashCode |= 0;  
    }  
    return hashCode;  
}
```

```
public asDataString(): string {  
    return this.getX() + "#" + this.getY();  
}
```

# Comparison Method

A comparison method is

- A query method that compares to objects on an ordinal scale

Example

- `compareDistance(other: Coordinate): number` // to origin

Prefixes

- None specifically

Naming

- Prefix + what is being compared

# Comparison Method Example

```
public compareDistance(other: Coordinate): number {  
    let thisR = Math.hypot(this.getX(), this.getY());  
    let otherR = Math.hypot(other.getX(), other.getY());  
    if (thisR == otherR) {  
        return 0;  
    } else if (thisR < otherR) {  
        return -1;  
    } else {  
        return 1;  
    }  
}
```

# Conversion Method [1]

A conversion method is

- A query method that returns a different representation of the object

Example

- `asDataString(): string`

Prefixes

- `as`, `to`

Naming

- Prefix + target type

# Conversion Method Example

```
public toString(): string {  
    return this.asDataString();  
}  
  
public asDataString(): string {  
    return this.getX() + "#" + this.getY();  
}
```



# The Ambiguous Semantics of toString(): string

---

Is it made for

- Humans (= human-readable representation of object) or
- Machines (= machine-readable representation of object)?

Is it used in an end-user UI, the debugger, or in a database?

# **3. Mutation Methods**

---

# Set Method

---

A set method is

- A mutation method that changes a logical field of the object

Example

- `setX(x: number): void`

Prefixes

- `set`

Naming

- Prefix + name of logical field

# Set Method Examples

```
public setX(x: number): void {
    this.x = x;
}

public setY(y: number): void {
    this.y = y;
}

public setR(r: number): void {
    let phi: number = Math.atan2(this.getY(), this.getX());
    this.setX(r * Math.cos(phi));
    this.setY(r * Math.sin(phi));
}

public setPhi(phi: number): void {
    let r: number = Math.hypot(this.getX(), this.getY());
    this.setX(r * Math.cos(phi));
    this.setY(r * Math.sin(phi));
}
```

# Command Method

A command method is

- A mutation method that makes a complex change to an object's state

Example

- `multiplyWith(other: Coordinate): void`

Prefixes

- `make`, `handle`, `execute`, `perform`, ...

Naming

- Prefix + descriptive term about the action

# Command Method Examples

```
public multiplyWith(other: Coordinate): void {  
    let newR = this.getR() * other.getR();  
    let newPhi = this.getPhi() + other.getPhi();  
    this.setR(newR);  
    this.setPhi(newPhi);  
}
```

# Remove vs. Delete

---

A remove command

- Removes an element from its context, but does not delete it

A delete command

- Deletes the element, invalidating any other references

# Initialization Method

An initialization method is

- A mutation method that sets some or all of the state of an object at once

Example

- `initialize(x: number, y: number): void`

Prefixes

- `init`, `initialize`

Naming

- Prefix + the part of the object being initialized



# Initialization Method Example

```
public initialize(x?: number, y?: number): void {  
    if (x !== undefined) {  
        this.x = x;  
    }  
  
    if (y !== undefined) {  
        this.y = y;  
    }  
}
```

## 4. Helper Methods

---

# Object Creation Methods

---

An object creation method is

- A helper method that creates an object and returns it

A factory method is

- An object creation method that creates an object by naming the class

A cloning method is

- An object creation method that creates an object by cloning an object

A trader (also: trading) method is

- An object creation method that creates an object from a specification

# Factory Method

---

An factory method method is

- An object creation method that creates an object by naming the class

Example

- `createCoordinate(x: number, y: number): Coordinate`

Prefixes

- `create` (also: `new`, `make`)

Naming

- Prefix + some identification for the new object

# Factory Method Example

---

```
public createOrigin(): Coordinate {  
    return new CartesianCoordinate(0, 0);  
}
```

# Cloning Method Example

---

```
export interface Cloneable {  
    clone(): Object;  
}  
  
public clone(): Coordinate {  
    return { ...this };  
}
```

# Naming Quiz

---

You need a method to create a new coordinate object. Which name is best?

1. `make():` Coordinate
2. `newCoordinate():` Coordinate
3. `createCoordinate():` Coordinate
4. `createNewCoordinate():` Coordinate

# Create vs. Ensure

---

A **create** command guarantees a new object

- `createLocationCoordinate(): Coordinate;`

An **ensure** command guarantees a specific cardinality of the requested object

- `ensureLocationCoordinate(): Coordinate;`

Create creates, ensure may or may not create a new object



# Assertion Method

---

An assertion method is

- A helper method that asserts a condition holds or throws an exception

Example

- `assertIsValidPhi(phi: number): void`

Prefixes

- `assert`

Naming

- Prefix + condition being asserted

# Assertion Method Example

```
public setPhi(phi: number): void {
    this.assertIsNotNullOrUndefined(phi);
    this.assertIsValidPhi(phi);
    this.phi = phi;
}

protected assertIsValidPhi(phi: number): void {
    if ((phi < 0) || (phi >= 2*Math.PI)) {
        throw new RangeError("Invalid phi value");
    }
}

protected assertIsNotNullOrUndefined(other: Object): void {
    if ((other == null) || (other == undefined)) {
        throw new RangeError("Value is null or undefined");
    }
}
```

## **5. Method Properties**

---

# Method Properties

---

A method property describes a particular property of a method

Method properties fall into different method property categories

A method may only have one property from any one category

Like method types, method properties have naming conventions

# Main Categories of Method Properties

---

## Method implementation properties

- Are properties of a method implementation

## Inheritance interface properties

- Are properties of the inheritance interface

## Convenience method properties

- Are about making programming easier

## Method meta-level properties

- Are about the method level (class, instance)

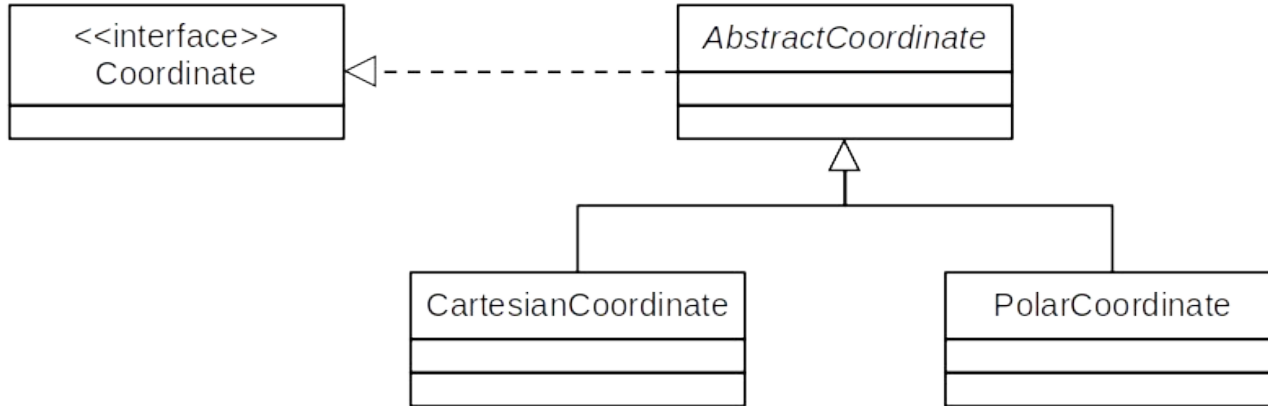
## Method visibility properties

- Are about visibility (public, protected, ...)

# Classification of Method Properties

Method implementation	Inheritance interface	Convenience
Regular	Regular	General
Composing	Template	Constructor
Primitive	Hook	Default value
Null	Abstract	Finalizer
...	...	...

# Extended Coordinate Example



# Coordinate Interface

```
import { Equality } from "./Equality";
import { Cloneable } from "./Cloneable";

export interface Coordinate extends Equality, Cloneable {
    reset(): void;

    getX(): number;
    setX(x: number): void;
    getY(): number;
    setY(y: number): void;

    calcStraightLineDistance(other: Coordinate): number;

    getR(): number;
    setR(r: number): void;
    getPhi(): number;
    setPhi(phi: number): void;

    calcGreatCircleDistance(other: Coordinate): number;
}
```



## **6. Method Implementation**

---

# Regular Method

A regular method is (just)

- A method that performs some task, for which it usually relies on further methods

Examples

- `initialize(x?: number, y?: number): void`

Prefixes

- None specifically

Naming

- None specifically

# Regular Method Examples

```
public isEqual(other: Coordinate): boolean {  
    return (this.getX() == other.getX()) && (this.getY() == other.getY());  
}
```

```
public calcStraightLineDistance(other: Coordinate): number {  
    let deltaX: number = Math.abs(other.getX() - this.getX());  
    let deltaY: number = Math.abs(other.getY() - this.getY());  
    return Math.hypot(deltaX, deltaY);  
}
```

# Composing Method

A composing method is

- A method that organizes a task into several subtasks as a linear succession of method calls to other regular or primitive methods

Examples

- `initialize(x?: number, y?: number): void`

Prefixes

- None specifically

Naming

- Adapted from Beck (1997)

# Composing Method Example

```
public initialize(x?: number, y?: number): void {  
    if (x !== undefined) {  
        this.setX(x);  
    }  
  
    if (y !== undefined) {  
        this.setY(y);  
    }  
}
```

# Primitive Method

A primitive method is

- A method that carries out one specific task, usually by directly engaging the object's implementation state; it does not use any non-primitive methods

Examples

- doSetX(x: number): void

Prefixes

- do, basic

Naming

- Prefix + name of the logical or implementation state

# Primitive Method Example

```
public setX(x: number): void {
    this.assertIsNotNullOrUndefined(x);
    this.doSetX(x);
}

protected doSetX(x: number): void {
    this.x = x;
}

public setPhi(phi: number): void {
    this.assertIsNotNullOrUndefined(phi);
    this.assertIsValidPhi(phi);

    let r: number = Math.hypot(this.getX(), this.getY());
    let x: number = r * Math.cos(phi);
    let y: number = r * Math.sin(phi);

    this.doSetX(x);
    this.doSetY(y);
}
```

# Null Method

---

A null method is

- A method with an empty implementation

Examples

- See Template Method example

Prefixes

- None specifically

Naming

- None specifically



## **7. Inheritance Properties**

---

# Template Method

A template method is

- A method that defines an algorithmic skeleton by breaking a task into subtasks the implementation of which is delegated to subclasses

Examples

- `Main.run()`

Prefixes

- None specifically

Naming

- Taken from Gamma et al. (1995)

# Template Method Example

```
export class Main {  
  
    public run(args: string[]): void {  
        this.parseArgs(args);  
        this.initialize();  
        this.execute();  
        this.finalize();  
    };  
  
    protected parseArgs(args: string[]): void {  
        // do nothing  
    }  
  
    protected initialize(): void {  
        // do nothing  
    }  
  
    protected execute(): void {  
        // do nothing  
    }  
  
    protected finalize(): void {  
        // do nothing  
    }  
  
}
```

# Hook Method

---

A hook method is

- A method that declares a well-defined task for overriding through subclasses

Examples

- `Main.parseArgs` / `initialize` / `execute` / `finalize`

Prefixes

- None specifically

Naming

- None specifically

## **8. Convenience Methods**

---

# Convenience Method

A convenience method is

- A method that simplifies the use of another, more complicated method by providing a simpler signature and by using default arguments

Examples

- `reset(): void`

Prefixes

- None specifically

Naming

- None specifically

# Convenience Method Example

---

```
public reset(): void {  
    this.initialize(0, 0);  
}
```

# Default-Value Method

A default-value method is

- A default-value method is a method that returns a single predefined value

Examples

- `static getOrigin(): Coordinate`

Prefixes

- None specifically

Naming

- None specifically, but typically also a getter



# Default-Value Method Example

---

```
public static getOrigin() : Coordinate {  
    return new CartesianCoordinate(0, 0);  
}
```

## **9. Design Guidelines**

---

# Pop Quiz!

Classify these methods

- clone(): Object
- equals(o: Object): boolean
- finalize(): void
- getClassName(): string
- getHashCode(): number
- notify(): void
- notifyAll(): void
- toString(): string
- wait(): void
- wait(timeout: number): void
- wait(timeout: number, nanos: number): void

# Single Method Purpose Rule

---

Single method purpose rule

- A method should have one purpose only

Benefits of single-purpose rule

- Makes methods easier to understand
- Makes overriding methods easier

# Exceptions to Single Purpose Rule

---

Well-known idioms

- Increment and return value (iteration)

Technical requirements

- Test and set value (critical sections)

Lazy initialization

# Documenting Method Types and Properties

---

Use annotations

# Homework

---

# Homework

- Implement the adap-b01 Name class as provided
- Use `string[]` as internal representation of name
- Annotate each method with its method type; example

```
// @methodtype get-method  
public getX(): number {  
    return this.x;  
}
```

- Commit homework by deadline to homework folder



# Summary

---

1. Method types
  - a. Query methods
  - b. Mutation methods
  - c. Helper methods
2. Method properties
  - a. Implementation related
  - b. Inheritance related
  - c. Convenience methods
3. Design guidelines

# Thank you! Any questions?

---

[dirk.riehle@fau.de](mailto:dirk.riehle@fau.de) – <https://oss.cs.fau.de>

[dirk@riehle.org](mailto:dirk@riehle.org) – <https://dirkriehle.com> – [@dirkriehle](#)

# Legal Notices

---

## License

- Licensed under the [CC BY 4.0 International](https://creativecommons.org/licenses/by/4.0/) license

## Copyright

- © 2012, 2018, 2024 Dirk Riehle, some rights reserved