Miniprojektiviikko

- Keskeisintä on oppia ymmärtämään periytyminen ja siihen liittyviä ominais/eriytymispiirteitä
  - Teoriatasolla myös SOLID-periaatteet
  - Teoriatason ymmärrys Design Pattern –termistä
    - Tästä lisää aineistoa keskiviikon/torstain aikana

- Varsinaisen ohjelmoinnin lisäksi
  - Kokeillaan git:n haastavampaa käyttöä
    - Branchit (haarat) ja niiden yhdistäminen
  - Noudatellaan (erittäin väljästi soveltaen) Scrum-projektityömallia

Miten työskennellään

- Joka päivä kaikki yhdessä virtuaalisessa luokkahuoneessa kello 9:00, 13:00 ja 16:00. Ensimmäinen sessio on vähän pidempi, muut lähinnä mahdollisuus esittää kysymyksiä
  - Muu aika peer-to-peer -huoneessa
- Jyrki on läsnä teams:ssä koko ajan
  - Ehkä satunnaisesti kurkkaa myös peer-to-peer -huoneisiin
- Lähtökohtaisesti koodataan omissa ryhmissä, mutta itsenäisesti….
  - Kuitenkin peer-to-peer huoneessa daily scrummit (tästä kohta lisää, mutta käytännössä tarkoittaa sitä, että juttelette keskenänne projektin etenemisestä)
    - Heti aamun ja iltapäivän aloittavan yhteisen session jälkeen sekä iltapäivällä noin kello 15

Vielä projektista

- Step 2, Filereporter (ja siitä eteenpäin)
  - Toteuttakaa piirre ensin omaan git branchiin (haaraan), jonka yhdistätte päähaaraan sen valmistuttua

- Mikäli haluatte lisää haastetta tehkää projektia varten oma github-repository, jonne viette ensin tyhjän projektin (“Terve maailma”
  - Molemmat kloonaavat projektin ja luovat omaa piirrettään varten branchin
  - Tämän jälkeen molemmat jäsenet työstävät omaa valittua piirrettään omissa brancheissään
    - Yksi: ScreenReporter-Person-PersonReport
    - Toinen: FileReporter-Company-CompanyReport
  - Kun molemmat valmistuvat ne yhdistetään päähaaraan

Ja vielä

- Tehkää projektista presentaatio

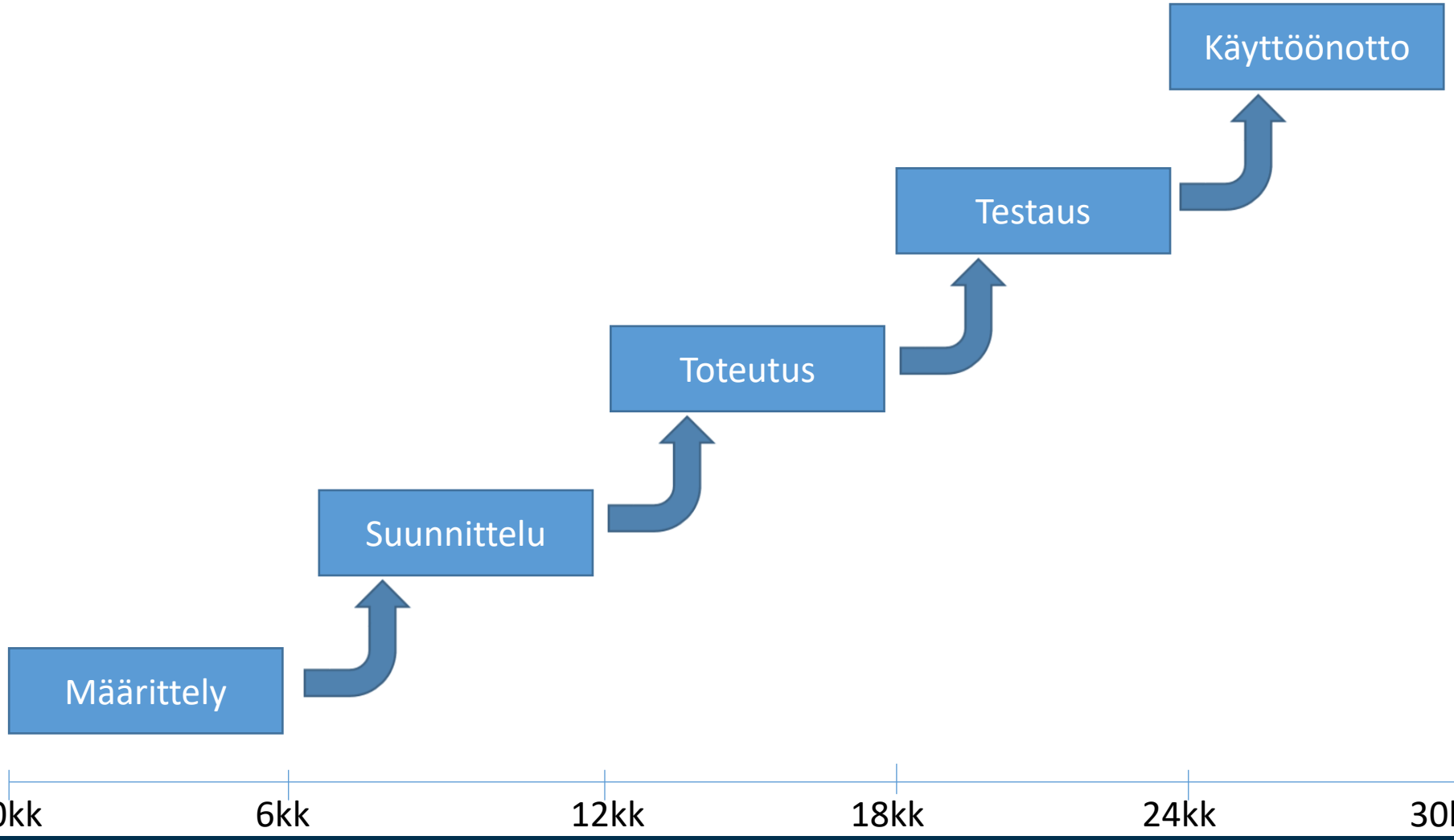
- Ensimmäiselle sivulle “product backlog” (lista piirteistä)
  - Projekti-ohje käytännössä kertoo piirteet, mutta aivan liian monisanaisesti
- Seuraavalle sivulle “luokkakaavio”
  - Laatikossa luokan nimi otsikkona ja sen alla listattuna tärkeimmät kentät ja metodit
  - Erilaisilla nuolilla kuvattuna luokkien väliset suhteet
    - Perii
    - Käyttää
    - Kurkkaa vaikka http://www.cse.hut.fi/fi/opinnot/CSE-A1121/2015/k01/osa03.html
  - Sen jälkeen kirjatkaa kunkin luokan vastuu
    - Sekä mitä muita SOLID-periaatteita sen toteutuksessa on käytetty
  - Ja sen jälkeen voitte vapaasti kirjoittaa ratkaisuun liittyviä omia muistiinpanoja liittyen kunkin luokan toteutukseen
    - Lähinnä tyyliin “Miksi tämä tehtiin näin”

Tiimit

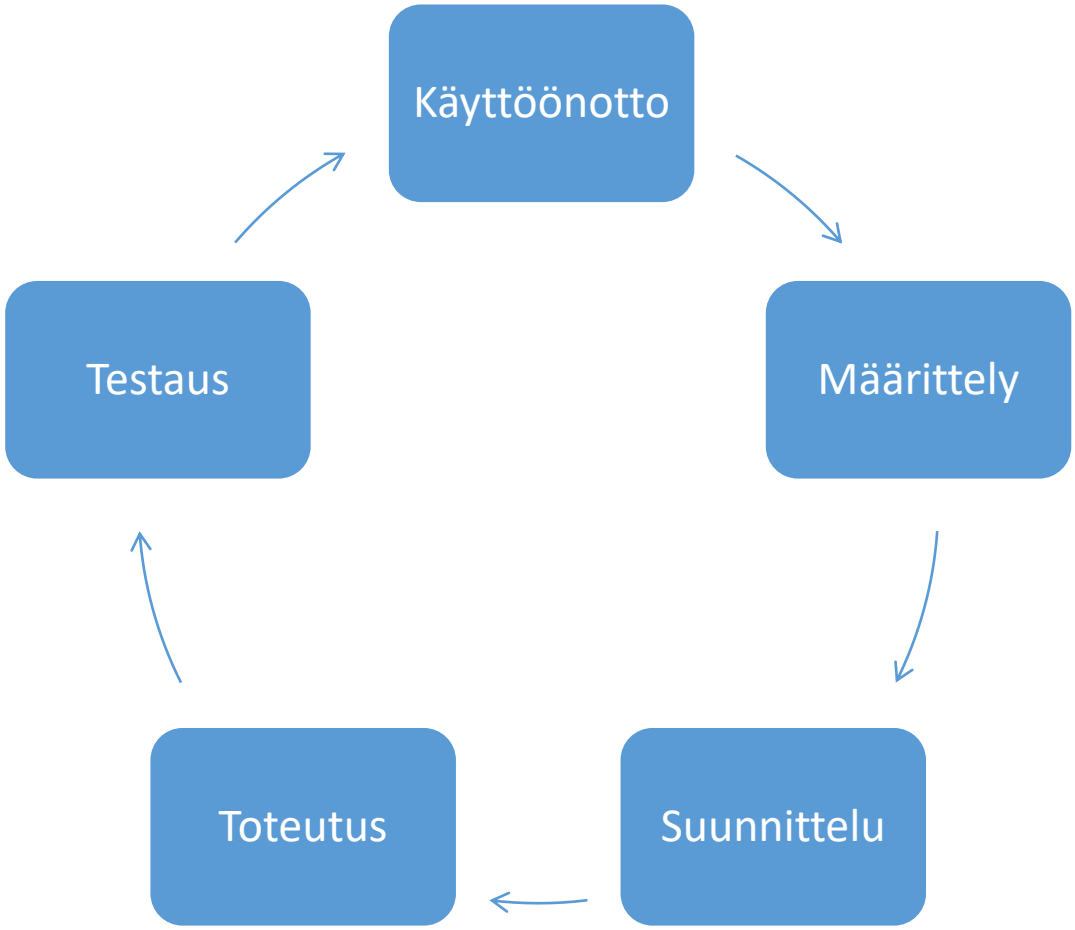
- Aapo-Mikko (peer-to-peer1)
- Henri-Susanna (peer-to-peer2)
- Kia-Kristian (peer-to-peer3)
- Kasperi-Taina (peer-to-peer4)
- Monica-Reijo (peer-to-peer5)
- Solja-Sakari (peer-to-peer6)
- Taito-Joel (peer-to-peer7)

Projektimalleja

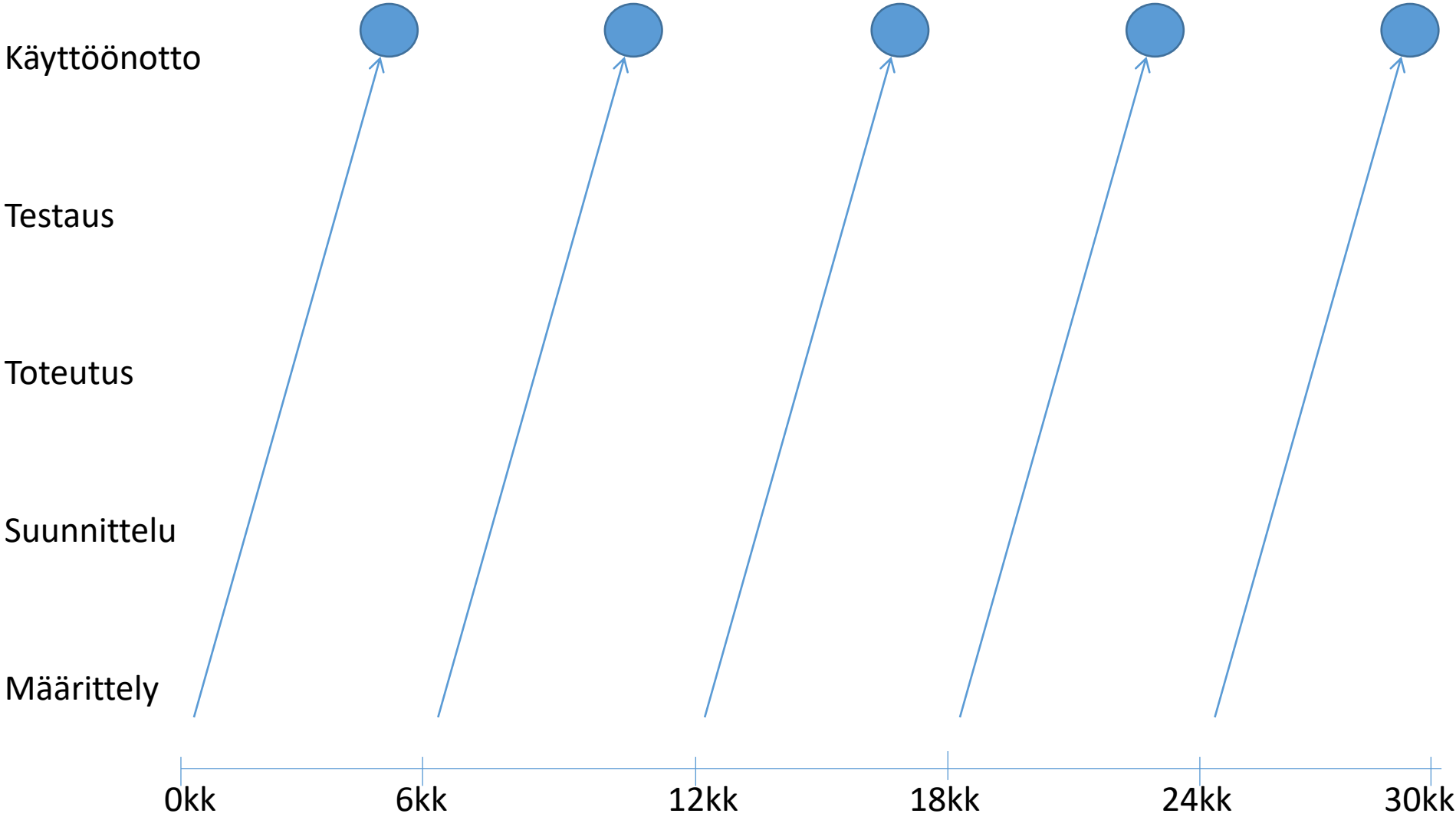
Vesiputousmalli



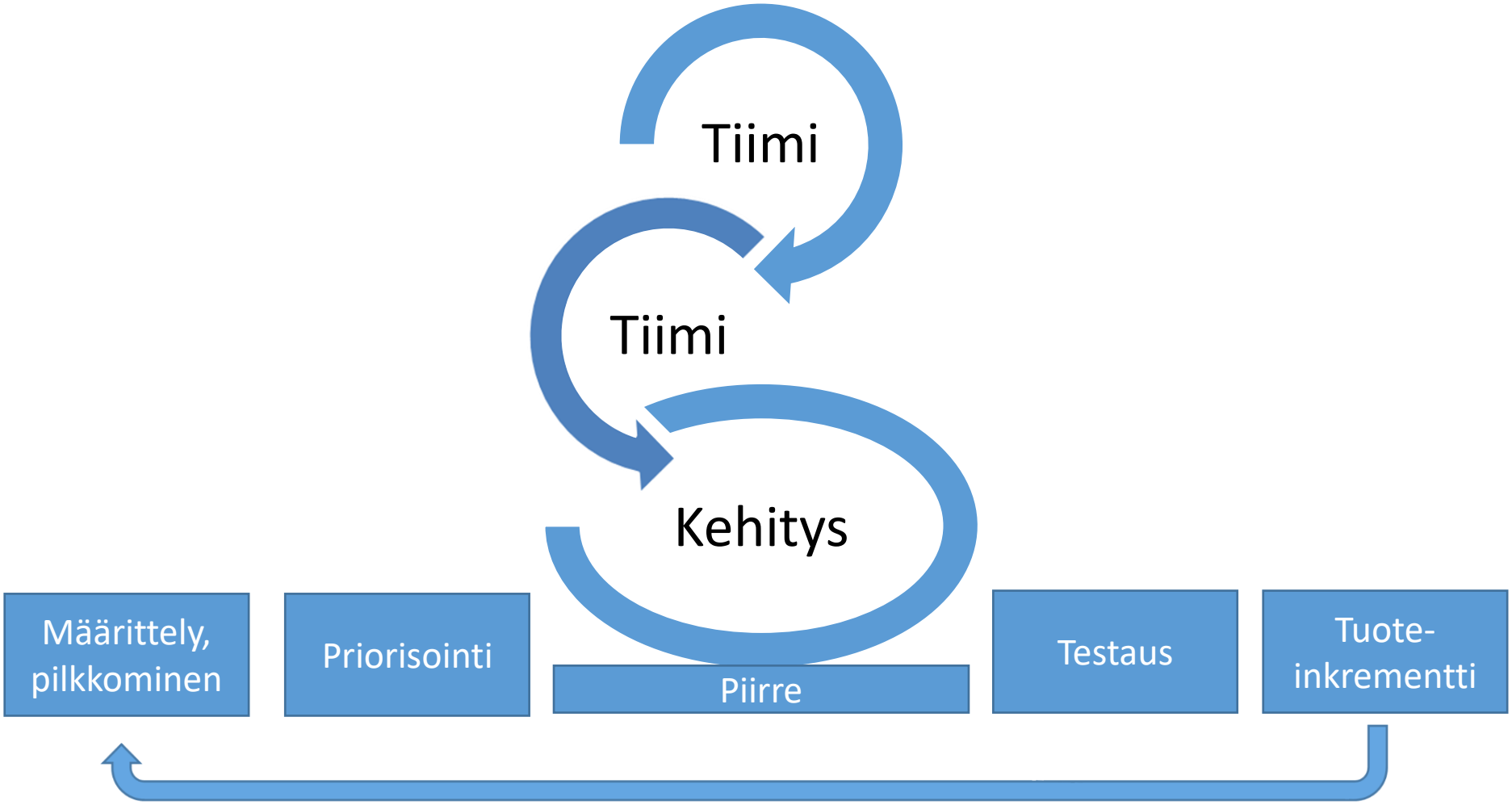
Iteratiivinen malli



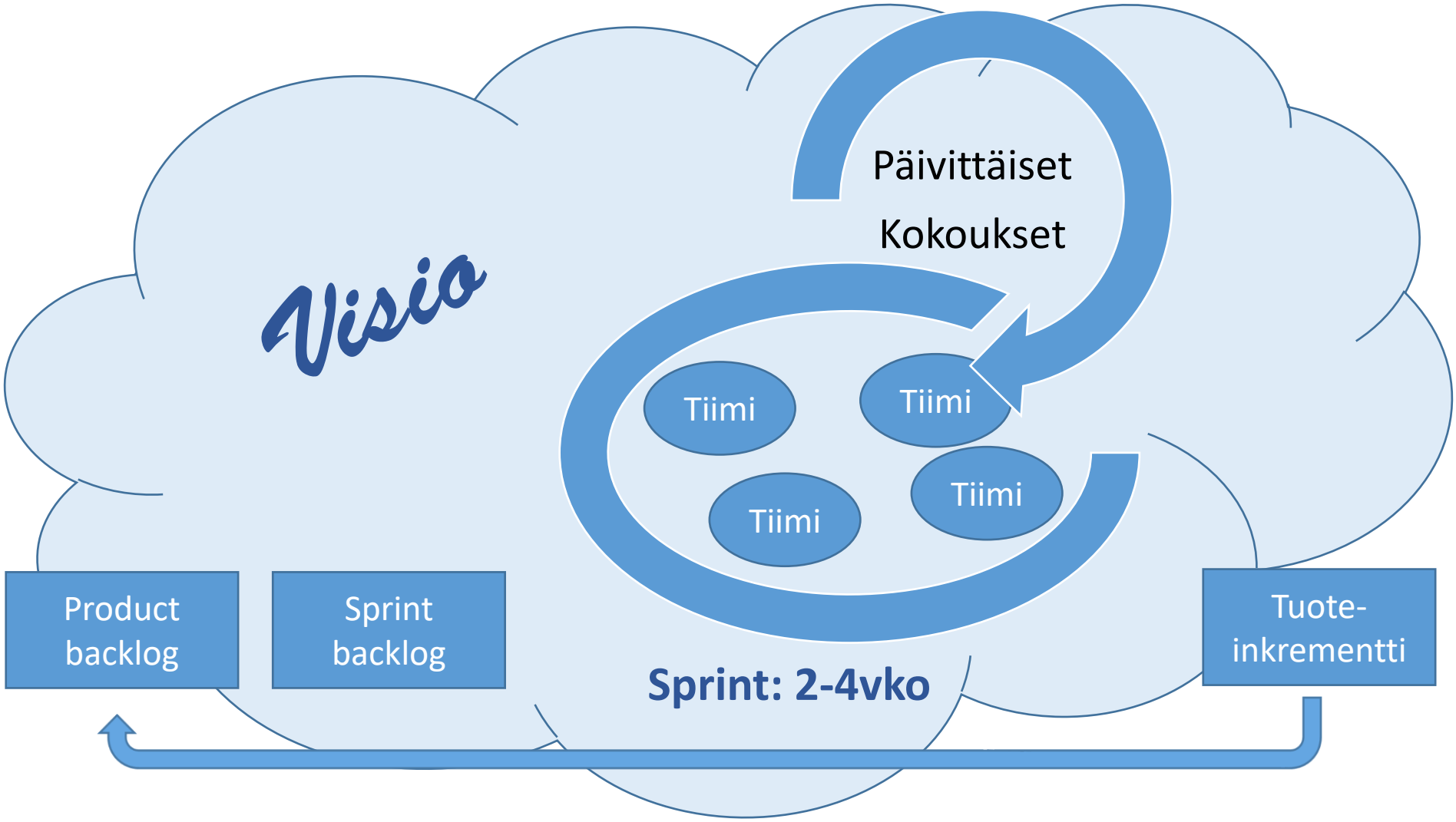
RAD – Rapid application development



Ketterä malli



Case Scrum


Visio
Päivittäiset
Kokoukset
Tiimi
Tiimi
Tiimi
Tiimi
Product backlog
Sprint backlog
Sprint: 2-4vko
Tuote-inkrementti

Agile Manifesto

- Muutokseen reagointi
  - Ennemmin kuin suunnitelman noudattaminen
- Asiakasyhteistyö
  - Ennemmin kuin sopimusneuvottelut
- Yksilöt ja vuorovaikutus
  - Ennemmin kuin prosessit ja työkalut
- Toimiva sovellus
  - Ennemmin kuin kattava dokumentaatio

Object oriented techniques

Why objects

- Why
  - Improved productivity through re-use
  - Improved maintainability
  - Safely extend and modify implementation
  - More precise responsibilities between pieces of source code

- How
  - Encapsulation (private fields, public getters and setters)
  - Inheritance (class extends another)
  - Polymorphism (methods with same name produce different results based on whom they are called)

How to design classes

- Most importantly who is responsible of what
  - What is the responsibility of a class
  - Who is responsible for the parameter validity
  - Hide complexities, ease of use
- What data is manipulated by objects
  - Who is responsible for the validity
  - Encapsulation
- How to modify and extend behaviour
  - ”Do not touch”
  - Inheritance and polymorphism

Solid

- Single Responsibility
  - Class should only have single responsibility, only one reason to change the implementation
  - Ensusing data validity and displaying the daa do not belong to the same class
- Open Closed
  - Entities should be closed for modifications but open for extensions
    1. Class can/should only be changed to correct errors
    2. Interfaces should describe behaviour, implementation can be changed
- Liskow Substitution
  - Object can always be references by a variable typed to any of the base classes
  - Types should be replaceable with their subtypes
  - Objects of subclass can always be used when an object of base class is needed
- Interface Segragation
  - Many client specific interfaces instead of one big interface
  - Design interfaces from client's (or service consumer's or object user's) point of view
    - What services are needed by the client
    - Use cases
- Dependency Inversion
  - Class that needs an object of a different class does not instantiate it but trusts that someone else gives it the object needed
  - Depend on abstractions instead of concretions (Interfaces instead of Class-types)
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details should depend on abstractions.

Classes

Roughly two types of classes

- For objects that hold data
  - Mostly contain publicly available data
  - Still data-validity should be responsibility of the object
    - Setters should contain logic
- For objects that provide services to manipulate data
  - Need for interface segregation?
  - Need for generics?

Basics of classes and objects

- You implement classes and instantiate them at runtime to get objects
- When instantiated the constructor for the object is called
  - Constructor initializes the contents (fields) of the object

```
public class Car {
    String model="";
    int numSeats=5;

    public Car(String m, int ns) {
        model=m;
        numSeats=ns;
    }
}
```

Variables declared at class-scope are called fields. All Car-objects will have a model and numSeats.

Constructor is called when the object is created. Two parameters need to be passed for constructor to be able to initialize the fields

```
Car volvo=new Car("V70",5);
Car renault=new Car("Twizy",2);
System.out.println(volvo.model);
System.out.println(renault.model);

Car vToo=volvo;
vToo.model="V40";
System.out.println(volvo.model);
System.out.println(vToo.model);
```

Two Car-objects are created. Variables volvo and renault are references to those objects

vToo refers to the same object as volvo. When model is changed with vToo, volvo's name changes also.

Objects

- Objects are created using reserved word 'new'
- Objects are not automatically copied
  - May be replaced and clone-method may be used
  - A copy-constructor may be written

  MyThing a = new MyThing();

  MyThing b = a;
- Object variables are references, unset member variable defaults to null

```
Car c=new Car(45);
```

Class members

Class may hold

- Fields
  - In most cases marked private so that they can only be accessed from the constructors and methods of the same class
- Constructors
  - Special 'method' that gets called when object is created using new-keyword
  - Method whose name matches the class name
  - No return value, not even void
  - Several versions may exist with different parameter lists
    - What information is needed to create a Car?
- Methods
  - Actual functionality provided by the objects
  - You always create an object and then ask it to do something, you call methods from objects.

Class members may be declared in any order

```
public class Car {
    private String model="";
    private int numSeats=5;
    private int speed=0;

    public Car(String m) {
        this(m,5);
    }

    public Car(String m, int ns) {
        this(m,ns,0);
    }

    public Car(String m, int ns, int s) {
        model=m;
        numSeats=ns;
        speed=s;
    }
}
```

This-keyword refers to the object whose method is being executed. Here you see a special use of calling another constructor.

Encapsulation (JavaBeans-model)

Classes that hold data should encapsulate it

```
public class Car {
    . . .
    private int speed=0;

    . . .

    public Car(String m, int ns, int s) {
        setModel(m);
        setNumSeats(ns);
        setSpeed(s);
    }

    public void accelerate(int ds) {
        setSpeed(speed+ds);
    }

    . . .

    public int getSpeed() {
        return speed;
    }

    public void setSpeed(int speed) {
        if ((speed>=0) && (speed<=120))
        this.speed = speed;
    }

}
```

Fields should be marked private

All changes go through "accessors"

Accessors define "Speed"-property

Verify data before applying changes

```
Car volvo=new Car("V70",5,80);
volvo.accelerate(80);
System.out.println("Speed is still "+volvo.getSpeed());
```

Class and class member visibility

- Class/Interface visibility:
  - public: visible to all inside the virtual machine
  - *default visibility (no keyword)*: package visibility (interfaces are public by default)
- Class member visibility:
  - public: visible to all
  - protected: visible inside the package and subclasses
  - default (no keyword): Visible to package and inside the same class
  - private: only visible inside the class

this-keyword

- This-keyword refers to the object from whom the method is called
  - Not available in static-methods
- Can be used to
  - separate member variable from a local variable (often parameter) with same name
  - As constructors first line to call another constructor contained in the same class
  - As a parameter to a method call indicating “who is making the call”

Static members

- Normal class members are object instance specific
  - Meaning each instance has their own set of independent values that may be manipulated through the instance reference (variable).
- In some cases classes require members that may be manipulated outside a specific instance, or have a generic field which is shared by all the instances
  - This "class-specific" member is marked using *static* keyword.
- Static class variables (fields) are used through the class name, not through the instance variable
- Static methods are also called using the class name instead of instance variable.

Static members, continued

- Static initializers may be used when initialization of static field requires complex logic
- Otherwise static members should be initialized when declared
- If you declare static member public it is pretty much global
  - Can be accessed (and changed) anywhere

```
public class Car {
    static int numTyres=4;
    static LocalDate justSample;

    static {
        String dateConf=Application.getConf();
        DateTimeFormatter fmt=DateTimeFormatter.ofPattern(dateConf);
        justSample=fmt.parse(Application.someData);
    }

    . . .
}
```

Static fields may be initialized when declared

If initialization is complex static initializer block may be defined

Object class methods

Object-class is the final base class for all Java-classes

- toString
- equals
- hashCode
- getClass
- finalize
- clone
- wait, notify

String-representation of object contents

Compare object state

Query type information

Destructor

Make a copy

Synchronization

toString() method

It’s not really required to implement this but

- If you implement it, it’s easier to do
  - Debug of log object state
- Implementation returns a String
- Typically you would return a String containing object name and enough of its state
- Default implementation returns object name and hashcode, separated by @

boolean equals(Object obj)

- Default implementation tests whether references point to the same object (==)
- This should be changed if you want to equals to compare contained information instead of object references
- Semantics may not be changed
    - No object equals null
    - symmetric: if a.equals(b), then b.equals(a)
    - transitive: if a.eq(b) and b.eq(c) then c.eq(a)
    - consistent: result may not change if objects remain the same
- Typically you:
    - Check that object is not null
    - Do a typecast (No need to worry about ClassCastException!)
    - Compare each field separately, between your object and parameter object. If they differ, return false. If all are same, return true.
- NOTE! Parameter should always be Object, not your own class type, otherwise you’re not really ovewriting!

int hashCode()

- On occasions equality of objects is tested
  - By first calling hashCode, if it returns different values equals is not called
  - If hashCode returns same values equals makes the final check
- When-ever equals is implemented, hashCode should be implemented also
- hashCode is a numeric value generated from object contents, that may be used to speed up object comparison and lookup
  - Pick one of the members where there is lot of variance between objects and return its hashCode
- By default hashCode is object memory address
  - Works OK with default implementation of equals

Object clone()

- Class needs to allow cloning by implementing Cloneable interface
  - Additionally, clone method may need to be overridden with public modifier instead of protected
- clone-method takes care of copying the object
  - Default implementation from Object class (super.clone()) is often good enough
- Clone may be overwritten for own objects
  - Protected member in the base-class
  - May throw CloneNotSupported –exception
  - The class must implement Cloneable

void finalize()

- When the garbage collector removes the object from the memory finalize is called for that object
- Finalize should not be used to release time-critical resources
  - Better to use own methods like close() or free() or release()
  - Finalize is just a backup
- Java application may end without ever visiting object finalize methods
  - Finalize should not be used to release limited resources

Inheritance

- We may define a separate base class for our class
  - Extends-keyword
- Class may only inherit a single class, multiple inheritance is not supported
  - And we always have a base class, if none is specified Object-class is used as the base class
- All base-class members are inherited to subclass
  - But we cannot access private members
- Super-keyword refers to the base-class
  - Constructors first statement may be a call to the base-class constructor
  - Use to refer to overridden member from base-class
- Extends…
  - Objects of inheriting class hold all the members or the base class
    - They can only access public and protected members
  - May add new field, accessors and methods

```
public class Bus extends Car {
    // Now Bus is exactly
    // the same as Car
}
```

Polymorfism

- We can always refer to the object of inheriting class with a variable that has type of base-class
  - Object o=new AnyTypeMayBeReferencedWithObject()
  - instanceof –operator also considers objects of inheriting class to also be instances of base-class
- Inheriting classes may contain members with same signature as the base class does
  - @Override annotation may/should be used to emphasize this
- If inheriting class overrides base class members, objects own implementation is always used
  - Result from method call depends on the object from whom it is called

Inheritance: Abstract class

- Occasionally you know in the base class that certain methods should be implemented into inheriting classes
  - Declare those methods in the base class without implementation
  - Mark the method with abstract-keyword
  - Mark the entire class also abstract
- Abstract class may not be instantiated, it must be extended and the extending class may be instantiated (If it's not abstract too)
  - Subclass must implement all abstract methods or it becomes abstract as well.
- Now you can fully benefit from polymorphism
  - Variable or parameter may be typed to the abstract class
  - We know that actual value is instance of subclass where the methods are actually implemented

```
abstract class Vehicle{
    protected int speed;

    public int getSpeed() {
        return speed;
    }

    abstract void setSpeed(int s);
}
```

Car extends Vehicle
- May check that the speed is between 0-120

Airplane extends Vehicle
- May check that the speed is between 0-1200

Interfaces, once more

Interface declaration

Anonymous classes

Functional interfaces

Lambdas

Method references

Inheritance: Interface

Up to JDK 1.7

- Interface just names methods that must be implemented elsewhere
  - Kind of completely abstract class, no method has an implementation
- Interfaces are not instantiated
  - No constructor
- Fields may be declared to an interface, but they are automatically “static final”
- Class promises to give implementation to the methods of an interface with “implements” -keyword

```
class UCaseFormatter implements Formatter {

    public String format(String data) {
        return data.toUpperCase();
    }
}
```

```
public interface Formatter {
    public String format(String data);
}
```

```
static void doPrint(String data, Formatter fmt) {
    System.out.println(fmt.format(data));
}

public static void main(String[] args) {
    Formatter ucase=new UCaseFormatter();
    doPrint("Hello world",ucase);

}
```

doPrint doesn’t know how to format

The caller decides

Anonymous classes

- Interfaces are not instantiated
- But you may instantiate an anonymous class that implements an interface
  - The notation looks like you would be instatiating an interface
- Very widely used feature
  - Learn to read and use

```
Formatter lcase=new Formatter() {
    public String format(String data) {
        return data.toLowerCase();
    }
};
doPrint("Hello world",lcase);

doPrint("Hello world",new Formatter() {
    public String format(String data) {
        return data.toUpperCase()+"!!!";
    }
});
```

lcase is an instance of anonymous class that implements formatter

Class is implemented here. Could hold other items also, but format-method is required

You may also use anonymous classes in this manner

Interfaces in Java 8

- Methods may also have static modifier
- Methods may also have implementation
  - Modifier ”default” must be used, you give default implementation, class may override
- In JDK 8 quite a few methods were added to existing familiar interfaces with either default or static modifier

```
public interface Formatter {
    public String format(String data);

    default public String pre() {
        return "";
    }
    default public String post() {
        return "";
    }
}
```

```
class HtmlFormatter implements Formatter{
    public String format(String s) {
        return pre()+s+post();
    }

    public String pre() {
        return "<p>";
    }

    public String post() {
        return "</p>";
    }
}
```

```
Formatter html=new HtmlFormatter();
System.out.println(html.pre()+"Hello world"+html.post());
```

Functional interface and lambdas

- Functional interface has only one abstract method
  - May have several methods with default implementation
- @FunctionalInterface may be used to emphasize the intent
- Lambda simplifies implementation of functional interface

```
doPrint("Hello world",new Formatter() {
    public String format(String data) {
        return data.toUpperCase()+"!!!";
    }
});
```

Using anonymous class

Using lambda

```
doPrint("Hello world",s -> s.toUpperCase()+"!!!");
```

Basically we just implement format-method: it takes the parameter s and produces s.toUpperCase()+”!!!” as the return value

First there were anonymous classes

- Use of anonymous classes is the traditional java-technique for callbacks

```
interface MyMath{
    int calc(int a,int b);
}


public class Demo {

    static void doCalc(MyMath t){
        System.out.println("Result: "+t.calc(2,3));
    }

    public static void main(String[] args) {
        doCalc(new MyMath(){
            public int calc(int a,int b){
                return a*b;
            }
        });
    }

}
```

Simple interface

Method taking object that implements the interface as parameter

Call method by instantiating an anonymous class that impmenlents the interface

Then came lambda-expressions (Java8)

- Shorthand for implementing anonymous class against an interface with just one method
  - Actually compiler doesn't generate the class file as it would in the sample on previous page

```
static void doCalc(MyMath t){
    System.out.println("Result: "+t.calc(2,3));
}

public static void main(String[] args) {
    doCalc((a,b) -> a*b);
    doCalc((a,b) -> a+b);
}
```

We must pass in an object that implements MyMath. It only holds calc-method so actually it has to be implemented.
We implement a method taking two parameters producing a return value from those.

The parameter list, compiler knows these are integers

The return value, again the compiler knows this should be an integer

When declaring the lambda

- You are always implementing a method declared in some interface
- You declare the parameter list
  - Compiler will know the types of parameters from the method declaration in the interface
  - A single parameter doesn't have to be surrounded by parenthesis
  - Parameter list of zero or two or more parameters must have the parenthesis
- And what is done with the parameters, actual method body
  - In most cases the body is just a single statement that can be evaluated into return value of the method.
  - If a block-is defined (multiple statements) the body also needs to contain actual return-statement
  - In the body you can use variables from "outer" scope but they must be final or effectively final

```
Runnable r = () -> System.out.println("Hello");
ActionListener l = e -> System.out.println(e.getActionCommand()) ;
Predicate<String> p = s -> !s.isEmpty();

MyMath c= (a,b) -> {
    if (a<=0) return b;
    if (b>100) return a;
    return a-b;
}
```

Functional interfaces, default methods

- To mark interface as one to be used with lambdas @FunctionalInterface - annotation may be used
  - The compiler will check that it will only contain one abstract methods
- In addition to single abstract method the functional interface may also contain methods with default implementation
  - A method marked with default-keyword can have an implementation defined for it in the interface definition

```
@FunctionalInterface
interface MyMath{
    int calc(int a,int b);

    default int calc3(int a, int b, int c){
        return calc(calc(a,b),c);
    }
}

// . . . . .
MyMath c= (a,b) -> {
    if (a<=0) return b;
    if (b>100) return a;
    return a-b;
};

System.out.println("Final "+c.calc3(1,2,3));
```

MyMath is almost like an abstract class. How-ever only interfaces can be used in association with lambdas.

Method with default implementation. There can be several of these.

Follow the logic to get the result of 3

Method references

- You may use method reference if you already have an existing method matching the signature of abstract method for functional interface

```
@FunctionalInterface
interface Processor{
    void process(String s);
}

public class Demo {

    static void doSomething(String s){
        System.out.println("Something:"+s);
    }

    static void takeProcessor(Processor p){
        p.process("Hello");
    }

    public static void main(String[] args) {
        takeProcessor(Demo::doSomething);  // Read as: s -> Demo.doSomething(s);
        takeProcessor(System.out::println);  // Read as: s -> System.out.println(s);
    }
}
```

Method references

- Method references may also be used to call method from the first parameter of interface method and instantiation

```
class Point{
    int x,y;

    public Point(int x,int y){
        this.x=x; this.y=y;
    }

    public String toString(){
        return "("+x+","+y+")";
    }
}

@FunctionalInterface
interface GeneratePoint{
    Point generate(int x,int y);
}

@FunctionalInterface
interface PointToString{
    String produce(Point s);
}
```

```
static void workWithPoint(
        int a, int b,
        GeneratePoint gp,
        PointToString ps){
    Point p=gp.generate(a, b);
    System.out.println(ps.produce(p));
}

public static void main(String[] args) {
    workWithPoint(4,5,
            Point::new,
            Point::toString);
```

Calls constructor

Calls p.toString()

Indicating errors

Throwing custom exceptions

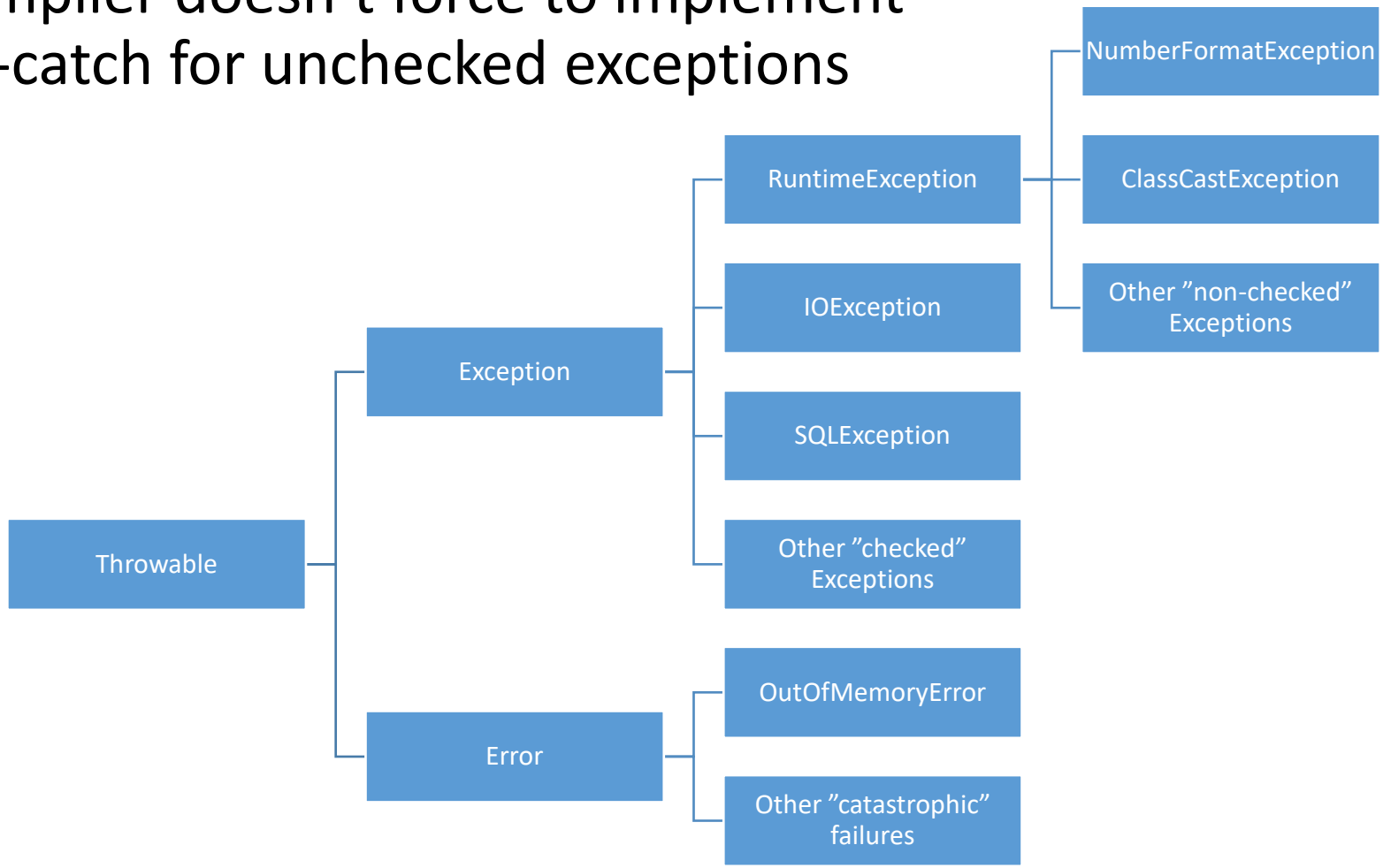
Reminder: catching exceptions

- If something goes wrong in the execution of your code an Exception is thrown
  - Exceptions may (and should) be caught in your code
- Exceptions are identified by their type
  - Class library provides you with huge number of exceptions
  - Documentations states which may be caused by different methods (IDE knows this also)
  - You may declare your own Exception types also (a class that extends Exception)
- You try to do something that may cause exceptions
  - Try block is followed by one or more catch-blocks, each processing specific type of exceptions
  - Last catch may just catch Exception, all exceptions listed before are caught
  - Optionally you may add finally block which is executed whether or not an exception is thrown in try block

```
String userInput="Tom,18"; // Name and age, comma separated
String[] arr=userInput.split(","); // Split the input into an array
try {
    int age=Integer.parseInt(arr[1]);  // Convert second item into int
}
catch(NumberFormatException ex) {
    System.out.println("Second item was not a number");
}
catch(Exception ex) {
    System.out.println("Something went wrong");
}
finally {
    System.out.println("Ready to continue");
}
```

Inheritance: Case exceptions

- Compiler doesn't force to implement try-catch for unchecked exceptions



Inheriting Exception

- Sometimes you want to indicate a logical error in your solution through exceptions
- You have option to pass extra data with self-implemented exceptions
  - Fields and suitable constructors

```
class MyException extends Exception{
    public String someRelevantData;

    public MyException(String data) {
        super("Something went wrong");
        someRelevantData=data;
    }
}

// Elsewhere
void exTest() throws MyException {
    . . .
    if (errorDetected) throw new MyException("Parameters are wrong");
    . . .
}
```