Introduction to Java programming

Object-oriented programming

Michiel Noback
Institute for Life Sciences and Technology
Hanze University of Applied Sciences

Object-oriented programming

- This presentation will deal with several aspects related to object-oriented programming (OOP):
  - Class design and implementation
  - Inheritance
  - Object construction
  - Access modifiers
  - Class Object
  - Interfaces, abstract classes and enums

PART 1: Basic OO principles

- In this part, we will get to know about ***inheritance & instance variables***
- We'll (re-)visit methods and how to ***override*** them
- We will have a look at ***access modifiers***: keywords used to specify the visibility (scope) of variables, classes and methods

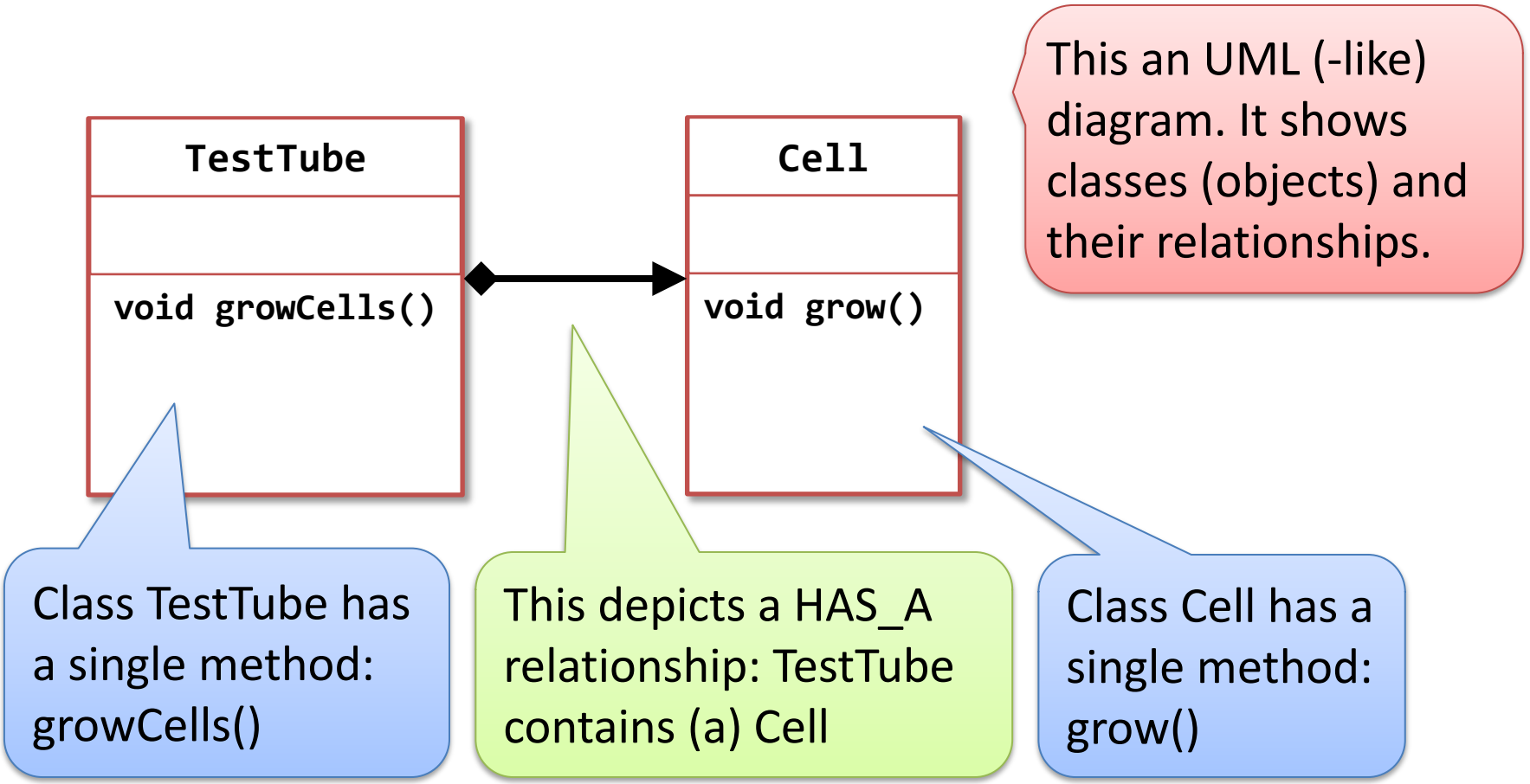
Programming a test tube

- The programming goal: build a general-purpose TestTube that is able to model the growth of all sorts of cells
- The starting material: A TestTube and a Cell class
- In UML it looks like this:

https://commons.wikimedia.org/wiki/File:Two_small_test_tubes_held_in_spring_clamps.jpg

Test tube in UML

Specifications of a general-purpose TestTube



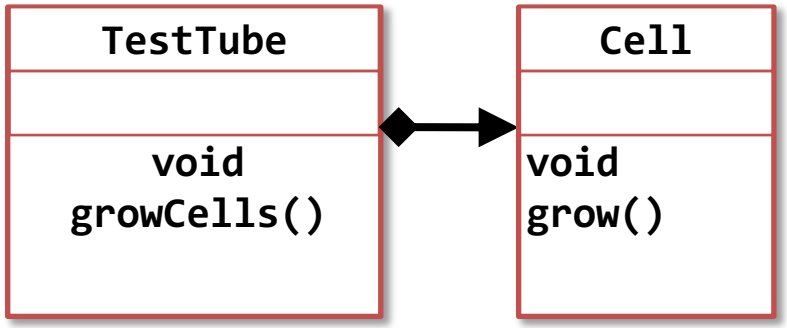
Code to go with the UML

```
public class TestTube{
    public static void main(String[] args) {
        TestTube testTube = new TestTube();
        testTube.growCells();
    }
    public void growCells() {
        Cell cell = new Cell;
        cell.grow();
    }
}
```

TestTube.java

```
public class Cell{
    public void grow() {
        System.out.println("Growing...");
    }
```

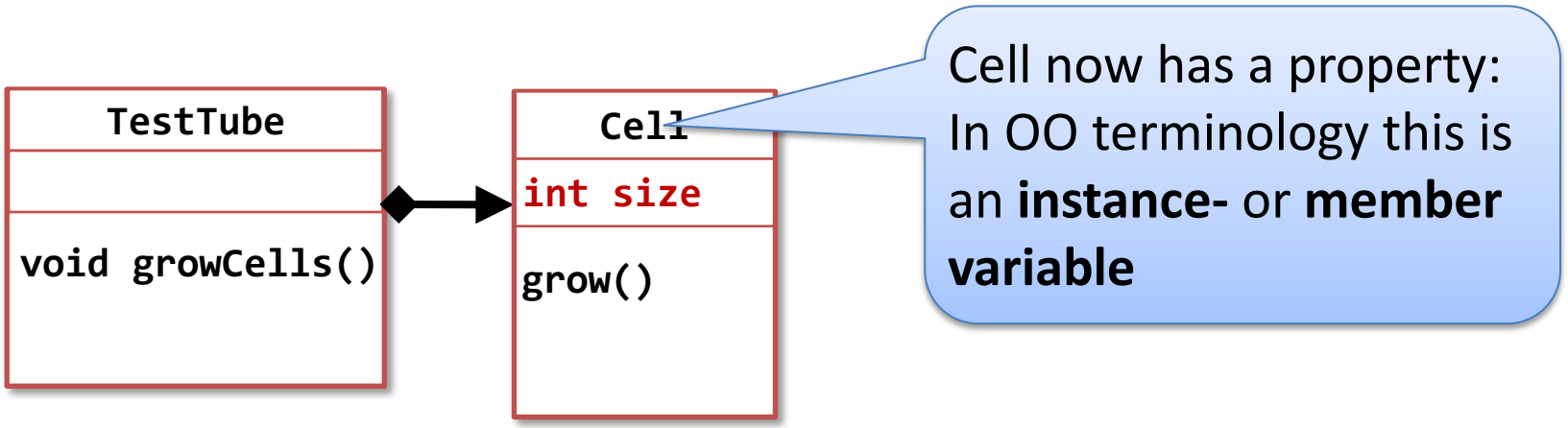
Cell.java



Introducing state

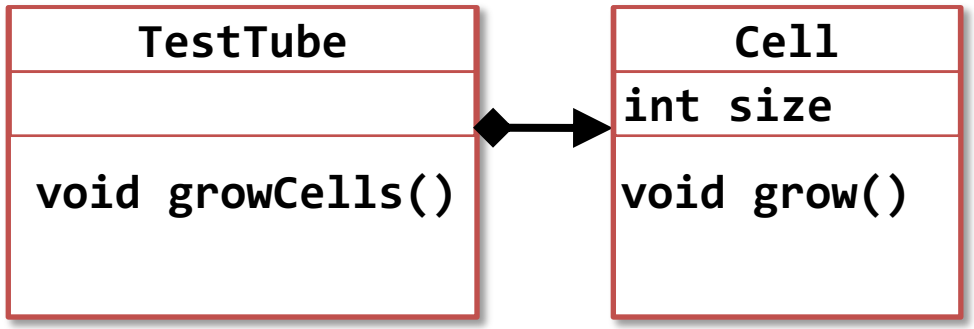
```
public class Cell{
    int size;

    public void grow(){
        System.out.println("Growing...");
    }
}
```



Instance variables and state

- An ***instance variable*** defines a property of an object.
- All instance variables together define the ***state*** of an object
- The state of an object usually is an important factor in how it will ***behave*** (how its methods execute)



Accessing state information (naively)

```
public class TestTube {
    ...

    public void growCells() {
        Cell cell = new Cell;
        cell.size = 14;
        cell.grow();
        System.out.println("Cell size is now" + cell.size);
    }
}
```

Give the cell a size...

...and fetch it later using the **dot operator**

```
public class Cell {
    int size;

    public void grow() {
        System.out.println("Growing...");
    }
}
```

Encapsulate that state!

Or: Getting to know each other….carefully!

```
public void growCells() {
    Cell cell = new Cell;
    cell.size = 24;
    cell.grow();
    System.out.println("Cell size is n
        + cell.size);
}
```

It may not be a good idea to let just anybody access ***and change*** your object state! Suppose Cell explosion size is 20 and TestTube does not know this?

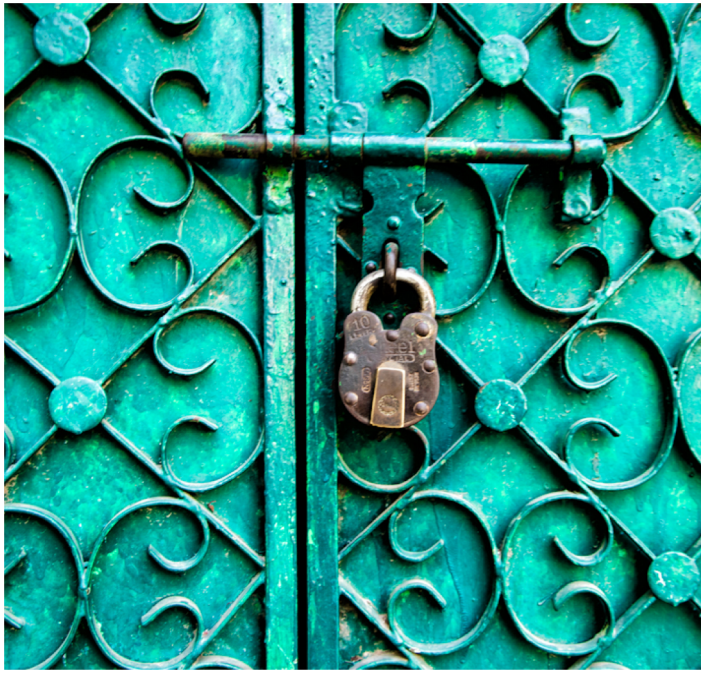
…or the size of the Cell is not for anybody to know?

"Keep it secret, keep it safe"

Meet our good friend encapsulation!

```
public class Cell{
    private int size;

    public int getSize(){
        return size;
    }
    ...
}
```



Making members **private** makes them inaccessible from outside your class code

https://commons.wikimedia.org/wiki/File:Lock_and_Door.jpg

"Keep it secret, keep it safe"

```
public class Cell{
    private int size;

    public int getSize() {
        return size;
    }
    public void grow() {
        System.out.println("Growing...");
    }
}
```

```
public void growCells() {
    Cell cell = new Cell;
    cell.size = 24;
    cell.grow();
    System.out.println("Cell size is now"
        + cell.getSize());
}
```

Will cause an error now! No permission

Politely run your request through the **getter** method

Encapsulation

- Encapsulation is a VERY important aspect of object-oriented programming
- It is central to
  - restricting and controlling access to object state
  - hiding knowledge about the inner workings of a class/object
  - making it possible to change implementation of functionality without affecting the rest of the code

Use getters and setters to intercept and control access

```
public class Cell {
    private int size;

    /*returns the cell's size*/
    public int getSize() {
        return size;
    }

    /*sets the size if within acceptable range*/
    public void setSize(int newSize) {
        if (newSize > 0 && newSize < 25) {
            this.size = newSize ;
        } else {
            throw new IllegalArgumentException("cell size outside
                    range 0-25");
        }
    }
```

Hide your variables! The **private** keyword ensures this

A **getter** that grants read access.

A **setter** that grants write access, but only within acceptable limits.

this ?! see next slide

The this keyword

- To indicate you want to access properties or methods of the current active object in class code, you use ***this*** together with the dot operator
- It is not required (the compiler will fill it in) to use ***this*** but it makes for much more readable code if you consistently use it when referring to object members (methods and variables)

```
this.size = 42;
```

Access modifiers

- Use access modifiers to control access (visibility) to classes, methods and variables
- Using no modifier is an (implicit) modifier, giving ***default*** - *but very distinct!* - behavior

Access modifier	***visibility***			
	Class	***Package***	***Subclass***	***All***
public	✔	✔	✔	✔
protected	✔	✔	✔	
default/no modifier	✔	✔		
private	✔			

Meet the professor

"…but how about het difference between bacteria and eukaryotic cells??

They are of way different size and grow at different speeds."

“I WANT TO GROW MANY TYPES OF CELLS IN MY TEST TUBE!!”

“And didn't I tell you? They should be able to divide as well.”

Thinking about the new specs

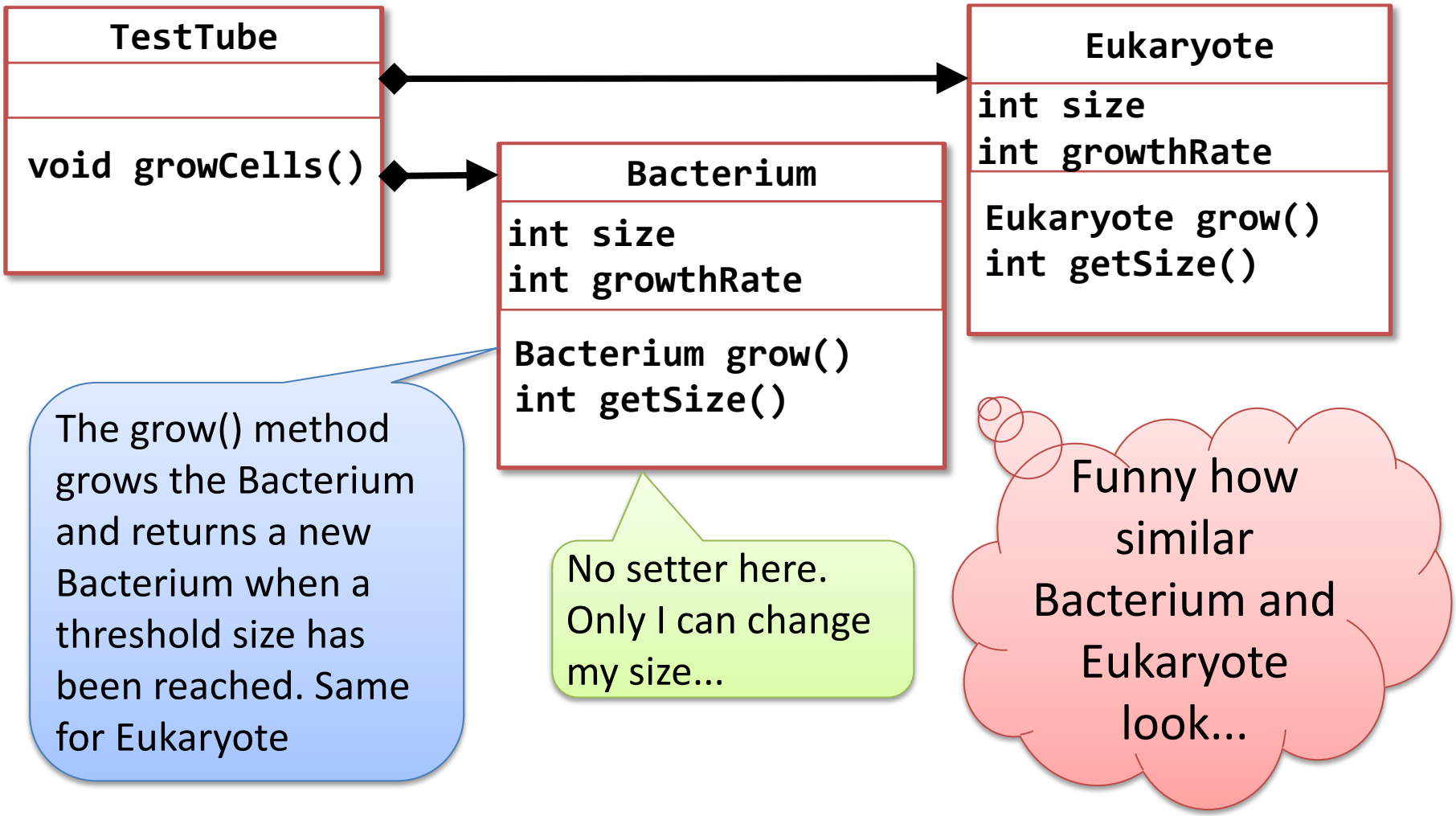
- When to divide?
- Who controls division:

  TestTube or the cell itself?
- How does this relate to growing?
- The different cell types is easy! Just give every type its own class.

Deciding how to implement the specs

I, as an expert on the subject, decide that the cells and not the test tube should know when to divide. Division is depending on the cell size: when a certain size is reached, it is time to divide.

Class diagram to model the new specs



Grow till you split

```
public class Bacterium{
    private int size = 10;
    private int maxSize = 25;

    /*returns the cell's size*/
    public int getSize(){
        return this.size;
    }
    /*growing functionality*/
    public Bacterium grow(){
        this.size += 5;
        if(this.size >= maxSize){
            this.size = 10;
            return new Bacterium();
        }
        return null;
    }
}
```

Calling **grow()** will increase this cell's size. If **maxSize** has been reached it will "divide" this cell and return a newly created one. If **maxSize** has not been reached it will simply return ***null***

Bacterium
int size int maxSize
Bacterium grow() int getSize()

Testtube babies

```
public class TestTube{
    public static void main(String[] args) {
        TestTube testTube = new TestTube();
        testTube.growCells("bacterium");
    }
    public void growCells(String type) {
        if (type.equals(“bacterium”)) {
            Bacterium bact = new Bacterium();
            Bacterium babyBact = bact.grow();
            //do more growing stuff
        } else if (type.equals(“eukaryote”)) {
            Eukaryote euk = new Eukaryote();
            Eukaryote babyEuk = euk.grow();
            //do more growing stuff
        }
        //room for other types such as Archaea,
        //or bone cells that will surely come
    }
```

Specify a type of cell to grow

Does this give you an uneasy feeling?

A family of cells

"…so, for each cell type I will have to create a new class with almost the same properties and methods?

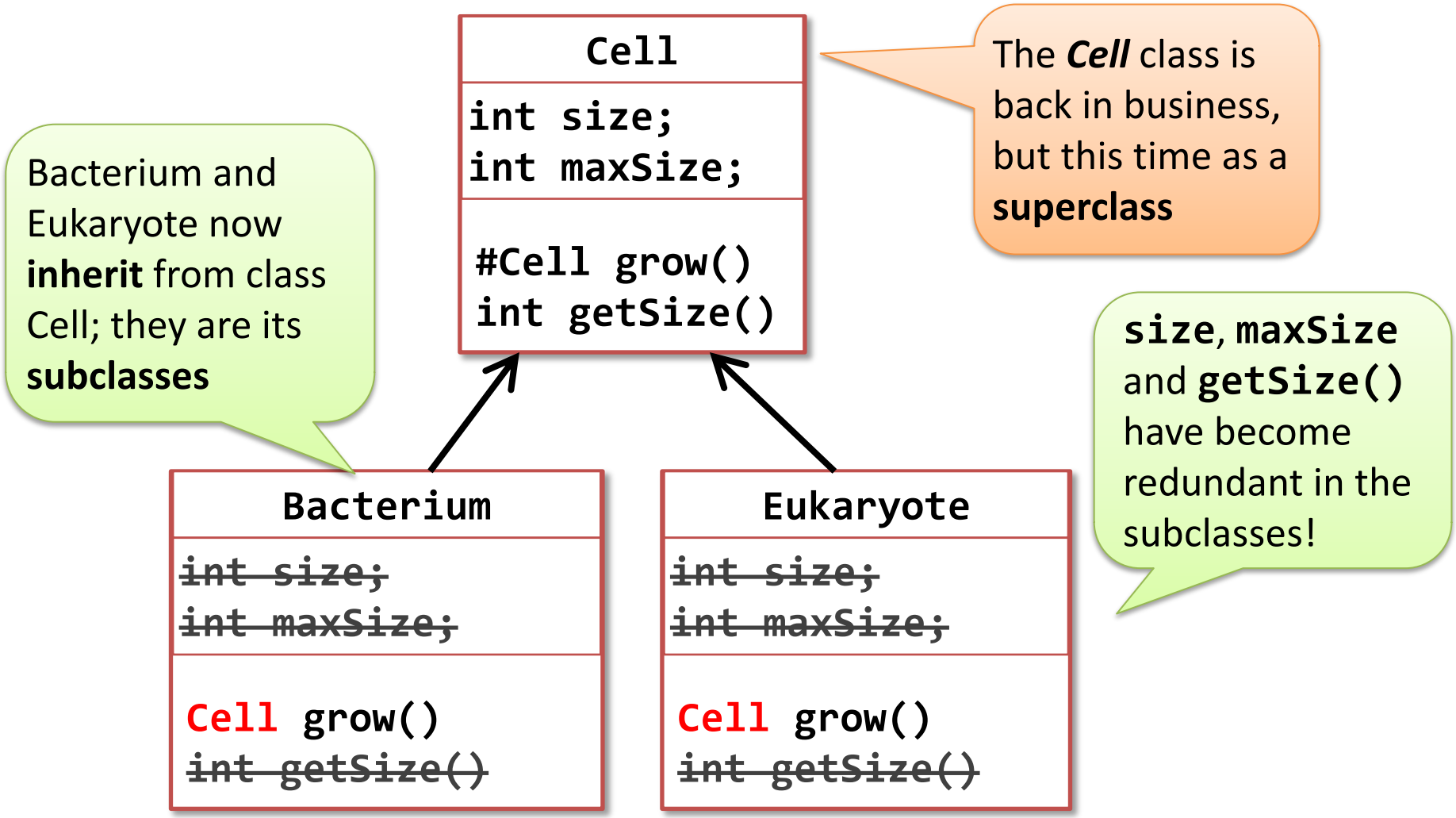
What a maintenance nightmare!"

I wish I could reuse code that is the same for each cell type and only code what is different.

BUT YOU CAN!!

Meet the wonderful world of inheritance!

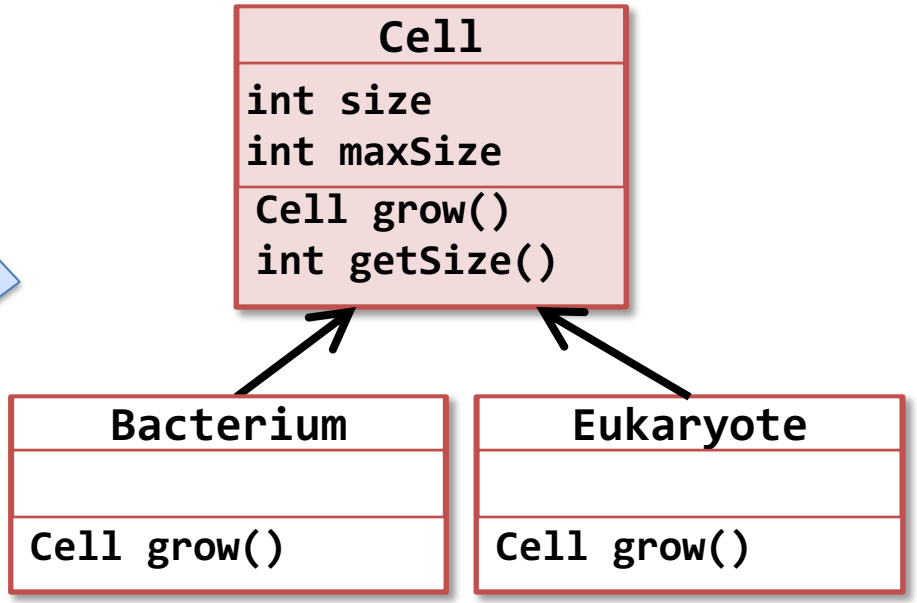
Inheritance: specifying family relations



Inheritance

- Class Cell defines what is applicable to all cells, whatever the specific type
- The ***subclasses*** will ***inherit*** these properties and methods as if they were their own

Cell defines what is in common. Bacterium and Eukaryote will only have to implement their own version of grow()



Inheritance: identify what is shared

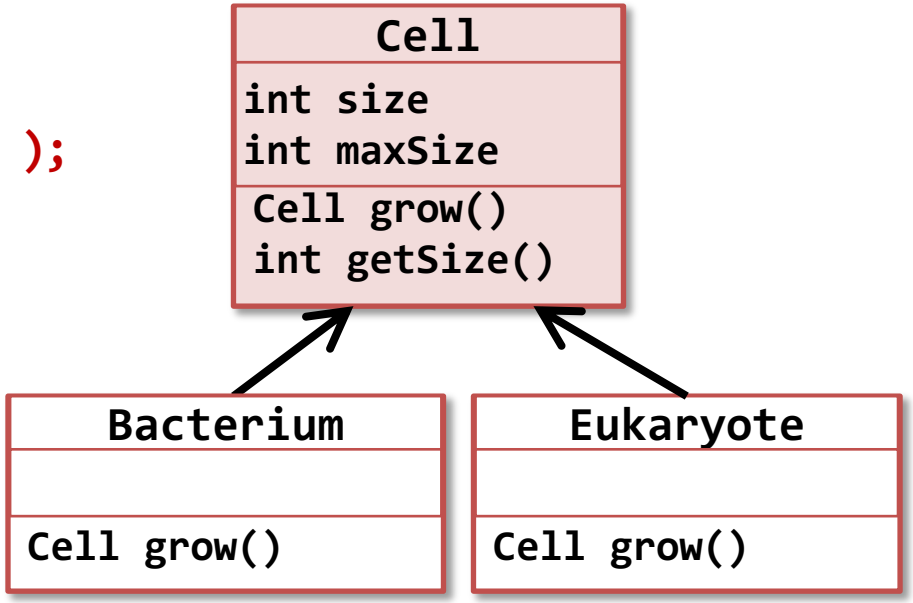
```
public class Cell{
    private int size = 10;
    private int maxSize = 25;

    /*returns the cell's size*/
    public int getSize() {
        return size;
    }
    /*growing functionality*/
    public Cell grow() {
        System.out.println(
            "don't know how to grow" );
        return null;
    }
}
```

Cell defines properties size and maxSize

Cell defines method getSize()

grow() should return a Cell, but how does a Cell grow? Therefore this dummy method. Can you think of a better solution?



Abstract classes

- Class Cell implemented a method - grow() - that had no logical default implementation; you simply cannot specify how a generic cell object should grow and divide.
- Therefore it is a good idea to make class Cell ***abstract***.

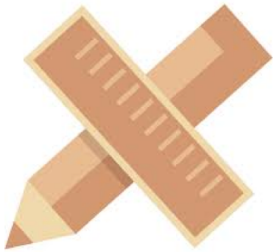
Abstract Cells

```
public abstract class Cell {
...
    /*returns the cell's size*/
    public int getSize() {
        return size;
    }
    /*growing functionality*/
    public abstract Cell grow();
}
```

Cell is made abstract: it is now impossible to code **Cell c = new Cell();**

Non-abstract methods are still allowed, providing implementations of functionality that are inherited by all subclasses.

grow() is made abstract: it has no method body: any concrete (non-abstract) subclass has to provide an implementation (has to override the method)



Design rule

- Make classes abstract
  - that should not be instantiated
  - that specify abstract methods - methods that have no logical default implementation and that concrete subclasses should therefore implement

Interfaces

- There is a special case of abstract class: when all its methods are declared abstract
- Such a class does not expose functionality, only a contract
- This special class type is called an **interface**, and you do not **extend** it but **implement** it
- We've already seen them when dealing with sorting
- In Java, you can implement many interfaces but only extend one class!

Inheritance begets simple subclasses

With the Cell class ready, now let's make a subclass:

```
public class Bacterium extends Cell {

    /*override Cell.grow()*/
    public Cell grow() {
       System.out.println("grow like a bacterium”);
         return null;
    }
}
```

extend Cell to indicate the inheritance relationship and get all Cell functionality

Bacterium has an obligation to implement grow() or it cannot be a concrete (non-abstract) class!

testing testing

```
public class TestTube5 {
  public static void main(String[] args) {
    //Cell cell = new Cell(); ← WILL NOT COMPILE
    //cell.grow();
    //System.out.println("cell size is " + cell.getSize());
    Bacterium bact = new Bacterium();
    bact.grow();
    System.out.println("bact size is " + bact.getSize());
  }
}
```

```
$ java TestTube5
grow like a bacterium
bact size is 10
```

Some real, functional code

Define method grow() to meet your idea of bacterial growth:

```
public class Bacterium extends Cell{
    /*override Cell.grow()*/
    public Cell grow(){
        System.out.println("grow like a bacterium");
        size += 5;
         if(size >= maxSize) {
             size = 10;
             return new Bacterium();
         }
         return null;
    }
}
```

size and maxSize are defined in class Cell, so I use them here as if my own. Just like the teacher said.

I cannot reach my inner self…

```
public class Bacterium extends Cell {
    /*override Cell.grow()*/
    public Cell grow() {
        System.out.println("grow like a bact
        size += 5;
         if (size >= maxSize) {
             size = 10;
             return new Bacterium();
         }
         return null;
    }
}
```

It won't compile!!!
What the f*** is going on here?

```
$ javac *.java
Exception in thread "main" java.lang.Error: Unresolved
compilation problems:
          The field Cell.size is not visible
          The field Cell.size is not visible
          The field Cell.maxSize is not visible
          The field Cell.size is not visible

          at Bacterium.grow(Bacterium.java:6)
          at TestTube5.main(TestTube5.java:9)
```

access access access

- What went wrong:

```
public class Cell{
   private int size = 10;
   private int maxSize = 25;
   //more Cell code
```

size and maxSize are private! This means visible to the own Class only! No access even for subclasses.
How to solve this?

access access access

- What can you do about it?
  - **Make them `public` again?**

size and maxSize are public again and accessible to anyone who wants to! We already decided that this is a **BAD IDEA**

```
public class Cell{
   public int size = 10;
   public int maxSize = 25;
   //more Cell code
```

http://sitr.com/holding-onto-old-data-really-really-bad-idea/

access access access

- What can you do about it?
  - Make them **protected**?

```
public class Cell{
   protected int size = 10;
   protected int maxSize = 25;
```

size and maxSize are now **protected**: visible to all classes in the chain of inheritance and/or in the same package

```
$ java TestTube5
grow like a bacterium
bact size is 15
```

```
Bacterium bact = new Bacterium();
bact.grow();
System.out.println("bact size is " +
                   bact.getSize());
```

access access access

- Make some **access methods**?

```
public class Cell{
  private int size = 10;
  private int maxSize = 25;
  /*read access to size*/
  public int getSize() {
    return size;
  }
  /*limited write access to size*/
  protected void setSize(int newSize) {
    size = newSize;
  }
  /*read access to maximum size*/
  public int getMaxSize() {
    return maxSize;
  }
}
```

size and maxSize are private, but accessible: setting the size from subclasses and getting size and maximum size from anywhere

There's no way to set the maxSize property but from within Cell.

Master, which is the right path to access?

So which is better, when achieving the same goal?

- **Using the protected keyword on members?**
- **Providing public getters and setters to grant access to private members?**
- **A combination?**

Each situation poses its own challenges and threats; meditate on it and the truth will reveal itself

(***...but in general, just use getters and setters***)

http://killbill.wikia.com/wiki/File:Pai_Mei_on_top_of_steps.jpg

Refactoring Bacterium

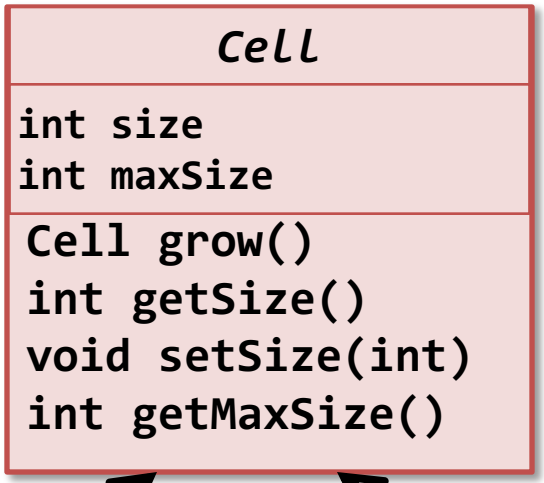
```
public class Bacterium extends Cell {

    /*override Cell.grow()*/
    public Cell grow() {
       System.out.println("grow like a bacteriu
       setSize(getSize() + 5);
        if(getSize() >= getMaxSize()) {
            setSize(10);
            return new Bacterium();
        }
        return null;
    }
```

Again, this compiles and runs OK!

```
$ java TestTube5
grow like a bacterium
bact size is 15
```

Inheritance summarized



Cell is the abstract supertype. It holds shared properties and methods. For grow behavior, it only provides a method signature that needs to be overriden.

Bacterium extends Cell and therefore only needs to implement grow(), which ***can return a Bacterium because it IS-A Cell***.

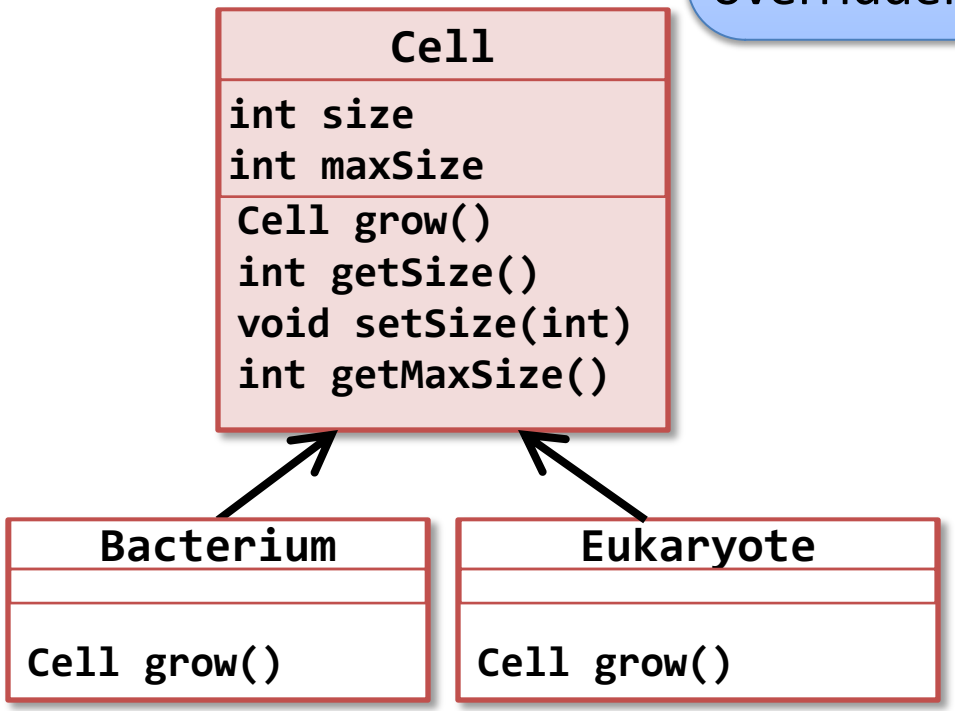
Same for Eukaryote.

So which method will be called?

```
Bacterium bact = new Bacterium();
Cell babyBact = bact.grow();
if(babyBact != null) {
   babyBact.getSize();
}
```

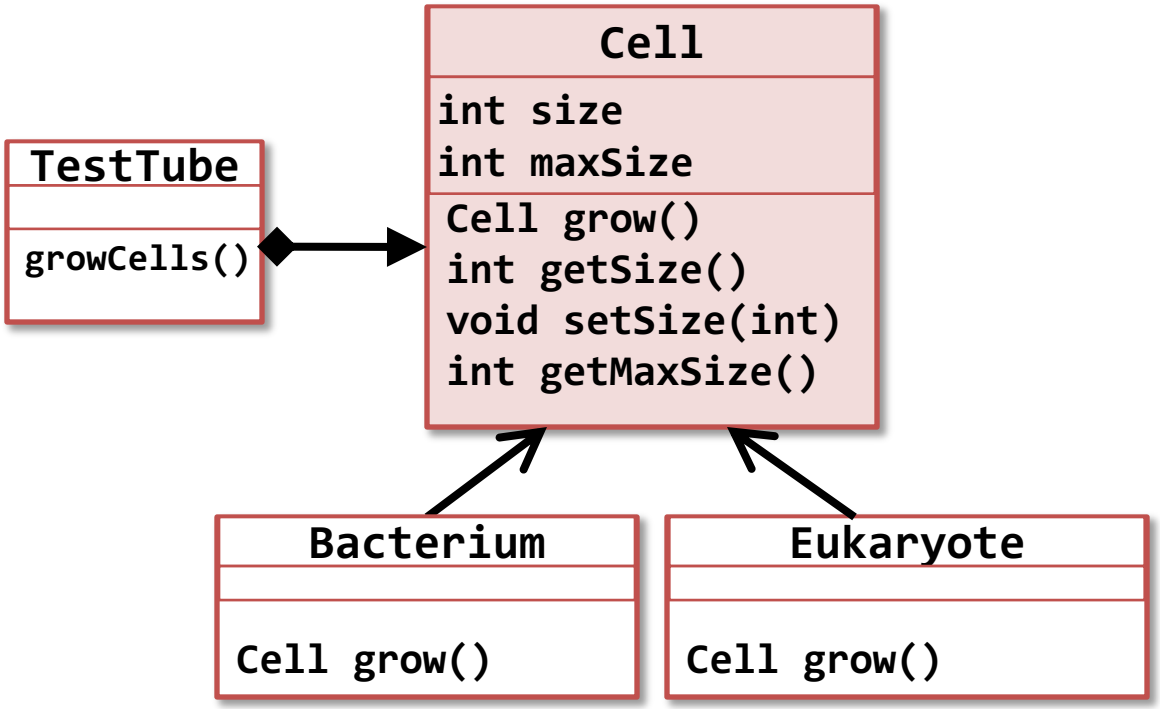
Calls grow() of class Bacterium because it is overridden

Calls getSize() of class Cell because it is NOT overriden



Inheritance

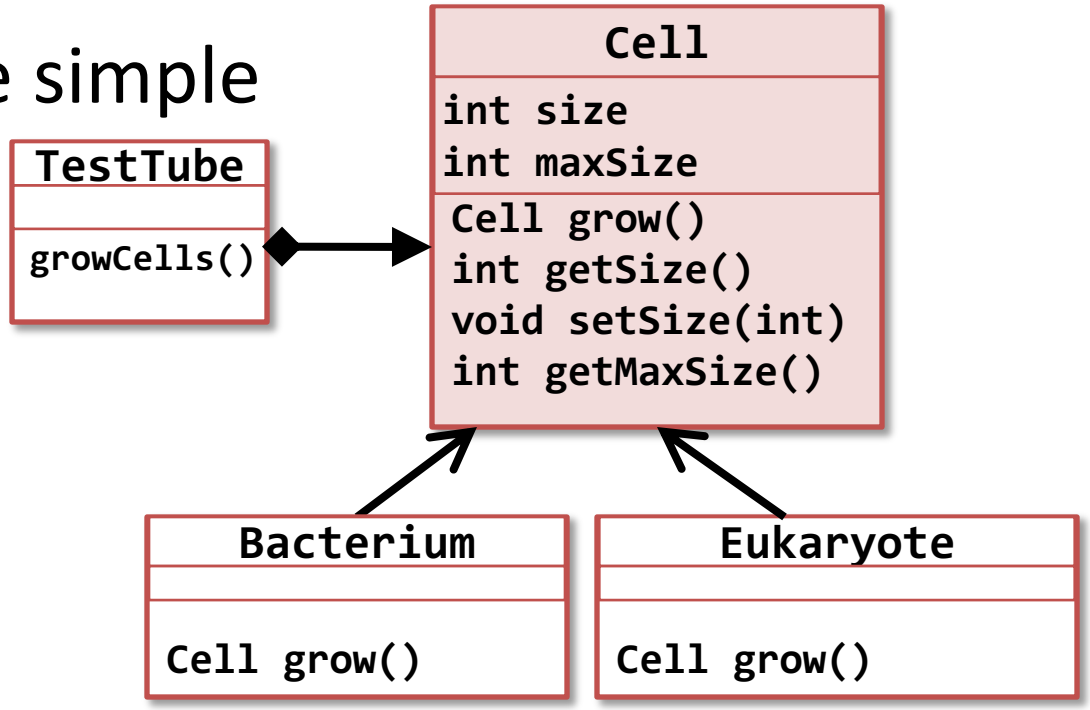
HOW DOES THIS HELP ME?



Inheritance

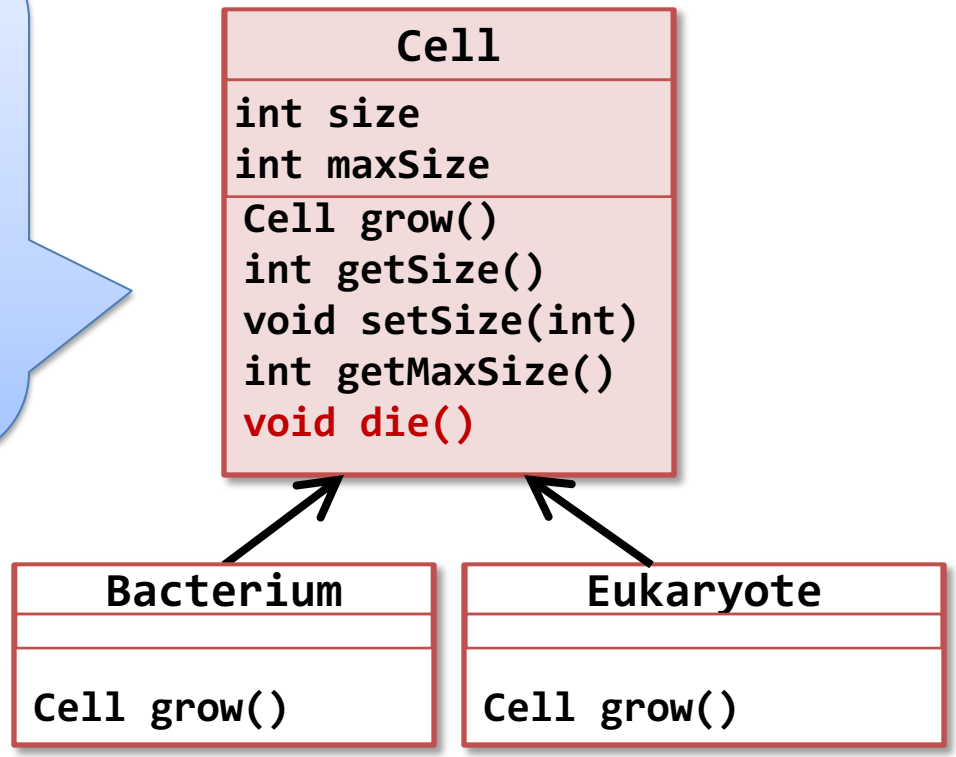
How does all this help me?

1. Keeping code from getting duplicated
2. Making the system flexible
3. Keeping TestTube simple



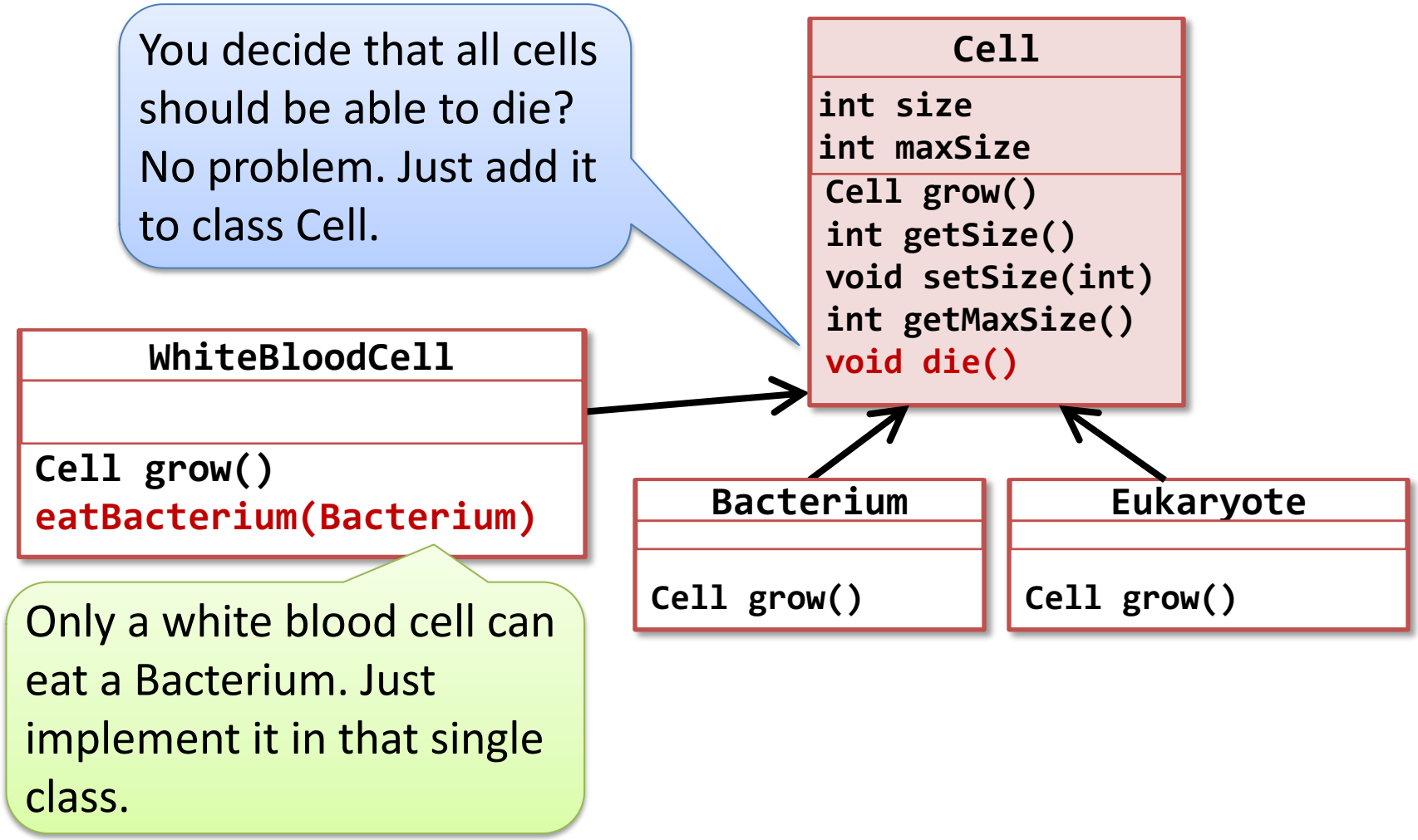
Keeping code form getting duplicated

All shared properties and functionality are defined in one place. Any code that needs to have these properties or functions will only have to subclass to achieve this.



Everything that is specific to a single class is defined in only that class. Like how to grow like a bacterium.

Flexibility



Keeping TestTube simple(r) through Polymorphism

```
public void growCells( String type ){
        Cell cell;
        if( type.equals("bacterium") ){
            cell = new Bacterium();
        }
        else if( type.equals("eukaryote") ){
            cell = new Eukaryote();
        }
        Cell babyCell = cell.grow();
    }
```

The generic type Cell is declared, but specific types are instantiated

The generic type is used to call the grow() method – and the correct implementation is executed

Object factories

```
public void growCells( String type ){
    Cell cell;
    if( type.equals(“bacterium”) ){
        cell = new Bacterium();
    }
    else if( type.equals(“eukaryote”) ){
        cell = new Eukaryote();
    }
    Cell babyCell = cell.grow();
}
```

Instantiating the correct type should actually be done elsewhere: in a factory method or class! A good topic for a next course.

Polymorphism

- Polymorphism allows one type to express some sort of contract, and for other types to implement that contract (often through class inheritance) in different ways.
- Code using that contract should not have to care about which implementation is involved, only that the contract is obeyed.

Summary OO

- In this part, you have met some of the essentials of object-oriented class design using inheritance
- In the next part, several aspects of object construction are explored

PART 2: Object construction

In this section we will look at

- the birth of objects - object construction and the methods that make this happen: **constructors**
- the Mother of all objects: **class Object**

Object construction

- An object is created (instantiated) when the constructor method of its class is called:

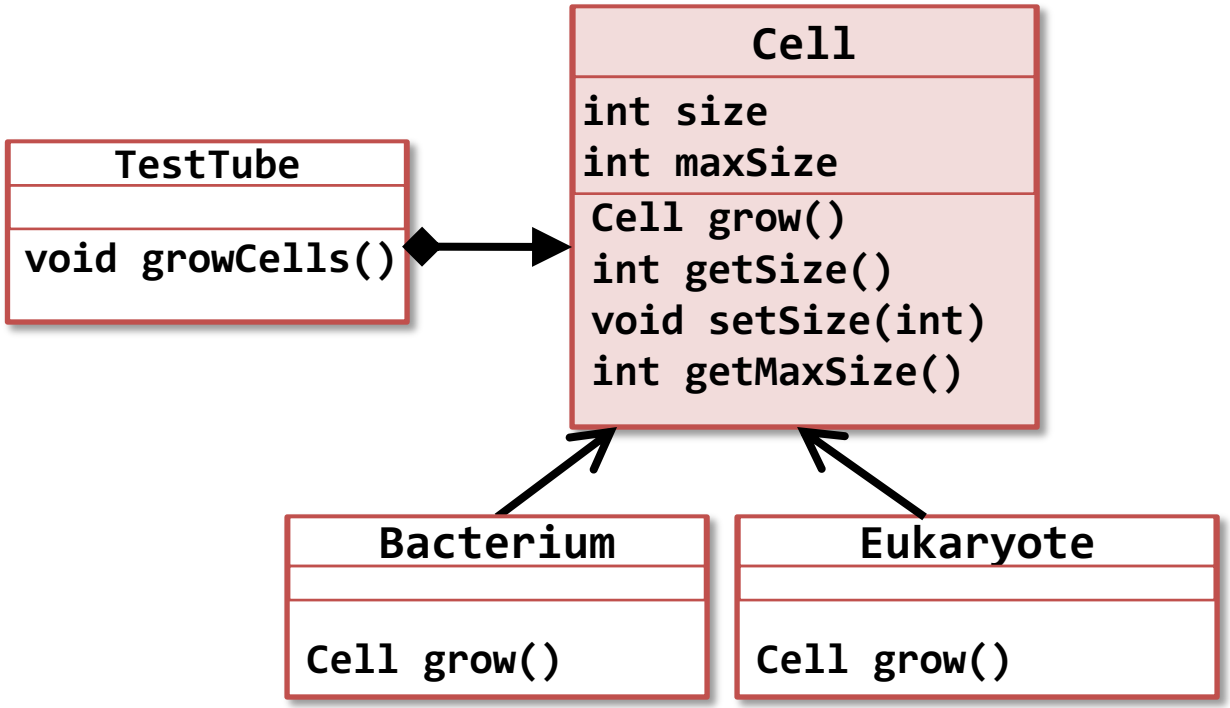
```
Cell cell = new Cell();
```

- But what happens here, exactly?
- To answer this we'll go back to the TestTube design, pimp it a little bit, and then look at Cell construction in detail

Class diagram

The TestTube class diagram

1. The class diagram includes a TestTube and Cell types.
2. TestTube contains an array of Cells that are all subclasses of class Cell.



Test Tube deluxe

We will run TestTube with the type and number of cells to grow specified from the command line

```
public class TestTube {
    public static void main(String[] args) {
        String type = args[0];
        int number = Integer.parseInt(args[1]);
        System.out.println("growing " + number
            + " cells of type " + type);
        TestTube testTube = new TestTube();
        testTube.growCells(type, number);
    }

    public void growCells(String type, int number) {
        //cell growing code
    }
```

```
$ java TestTube bacterium 100
growing 100 cells of type bacterium
```

Test Tube deluxe

```
public class TestTube {
   public static void main(String[] args) {
      //startup code
      testTube.growCells(type, number);
   }
     public void growCells(String type, int number) {
        Cell[] cells = new Cell[number];
        if (type.equals("bacterium")) {
           for (int i = 0; i < number; i++) {
               cells[i] = new Bacterium();
           }
        }
        else if (type.equals("eukaryote")) {
          //eukaryote code
        }
        //growing code
     }
```

An array of supertype Cell is created of required length and then filled with cells of the correct type

A simple cell class

```
public class Cell {
    private int size;

    /*returns the cell's size*/
    public int getSize() {
        return size;
    }
    /*dummy method!*/
    public Cell grow() {
        System.out.println("doing nothing");
    }
}
```

To investigate object construction, we go back to the simple times, when we only had a single type of cell

Phases of cell construction

- The three steps of object declaration, creation and assignment:

```
Cell myCell = new Cell();
```

(1) **declare** a reference variable

```
Cell myCell = new Cell();
```

(2) create / **construct** an object

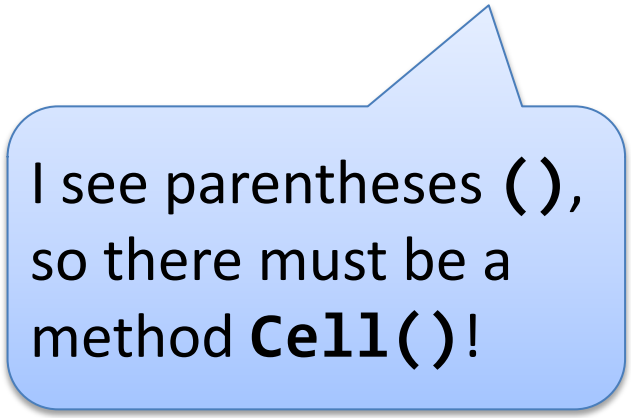
```
Cell myCell = new Cell();
```

(3) **assignment**: link the object and the reference

Cell construction

- So, are we calling a method named Cell()?

```
Cell myCell = new Cell();
```



Cell construction

- What **does** happen with cell construction?

```
Cell myCell = new Cell();
```

```
public class Cell {
    private int size;

    /*returns the cell's size*/
    public int getSize() {
        return size;
    }
    /*dummy method!*/
    public Cell grow() {
        System.out.println("doing nothing");
    }
}
```

I do not see a method **Cell()**, or **new()** for that matter.
Class Cell does not **have** a constructor method!

meet invisible code

- In a class without any constructor code, you get this for free:

```
public class Cell{

    public Cell(){ }

}
```

The compiler creates this stub for you, automatically

https://pixabay.com/en/invisible-man-spy-agent-anonymous-154567/

The default constructor

```
public class Cell{

    public Cell(){ }

}
```

Why should I be happy with this method? It does not take arguments, it does not do anything, nor does it return anything!

Meet invisible code

```
public class Cell{

    public Cell(){ }

}
```

..but it really DOES something: it creates a Cell object. And it DOES return something: a reference to the just created Cell object !

Initializing default object properties

```
public class Cell{
    private int size;
    //more Cell code
}
```

When a Cell object is created, it will have a `size` property set to the **default value** of its corresponding type (0 for an int)

```
public class CellTest{
   public static void main(String[] args){
      Cell myCell = new Cell();

      System.out.println( “Cell size is ” + myCell.getSize() );
   }
}
```

Initializing objects properties
The simple solution

```
public class Cell{
    private int size = 10;
    //more Cell code
}
```

When a Cell object is created, it will **always** have size 10

Initializing object properties
The flexible solution

```
public class Cell {
    private int size;

    public Cell(int cellSize) {
        this.size = cellSize;
    }

    //more Cell code
}
```

Define a **non-default** constructor. Now you can be **sure** the size of the cell will always be set when it is created, because arguments MUST be matched

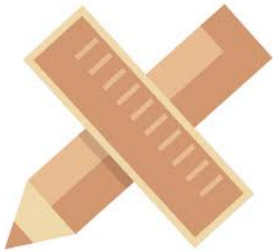
Matching constructor arguments is not optional

```
public class Cell {
    public Cell(int cellSize) {
        this.size = cellSize;
    }
}

public class CellTest{
    public static void main(String[] args) {
        Cell myCell = new Cell();
        System.out.println("Cell size is " + myCell.getSize());
    }
}
```

Now if you try to create a cell without specifying its size, you get a compilation error!

```
$ javac CellTest
Exception in thread "main" java.lang.Error: Unresolved
compilation problem:
        The constructor Cell() is undefined
        at CellConstrTester.main(CellConstrTester.java:9)
```



Design rule

- When a class defines an instance variable that needs to be initialized in order to have an object that makes sense, and you can not give it a reasonable default value, you should make it a constructor parameter

Multiple constructors

- You are allowed to create multiple constructors. For instance, if you want to make maxSize optional as well, you could use this set of constructors

The default maxSize is 30

```
public class Cell {
    private int size;
    private int maxSize = 30;

    public Cell(int cellSize) {
        this.size = cellSize;
    }
    public Cell(int cellSize, int maxSize) {
        this.size = cellSize;
        this.maxSize = maxSize;
    }
}
```

Now you can create Cell instances in these two ways:

```
new Cell(10);
new Cell(10, 50);
```

Multiple constructors

- The preferred this way of implementing multiple constructors is this

```
public class Cell {
    private int size;
    private int maxSize;

    public Cell(int cellSize) {
        this(cellSize, 30);
    }

    public Cell(int cellSize, int maxSize) {
        this.size = cellSize;
        this.maxSize = maxSize;
    }
}
```

Calls the second constructor with the default value

Object construction

- Actually, Java has one constructor that applies to all classes: the Object constructor.
- That's right, there is a class Object!!
- It is a very very special class: it is the ancestor of **all** Java classes.
- All Java classes inherit from class Object

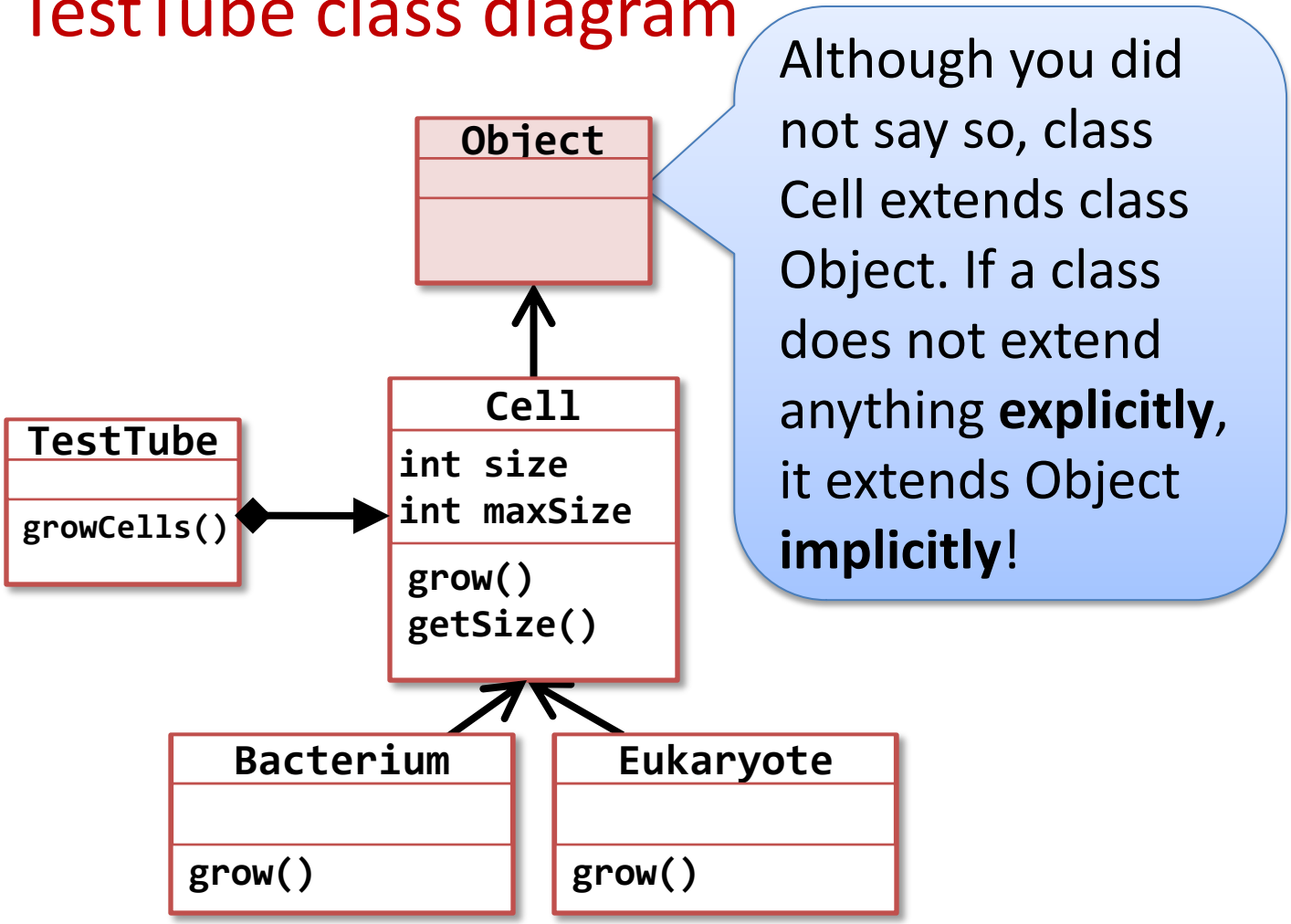
```
public class Bacterium extends Cell{ }

public class Cell extends Object{ }
```

Again code that the compiler provides for you, for free (but you are allowed to type it yourself)

Class diagram

The actual TestTube class diagram



Class object

- So, what is in class object?

The mother of all objects is instantiated just like that!

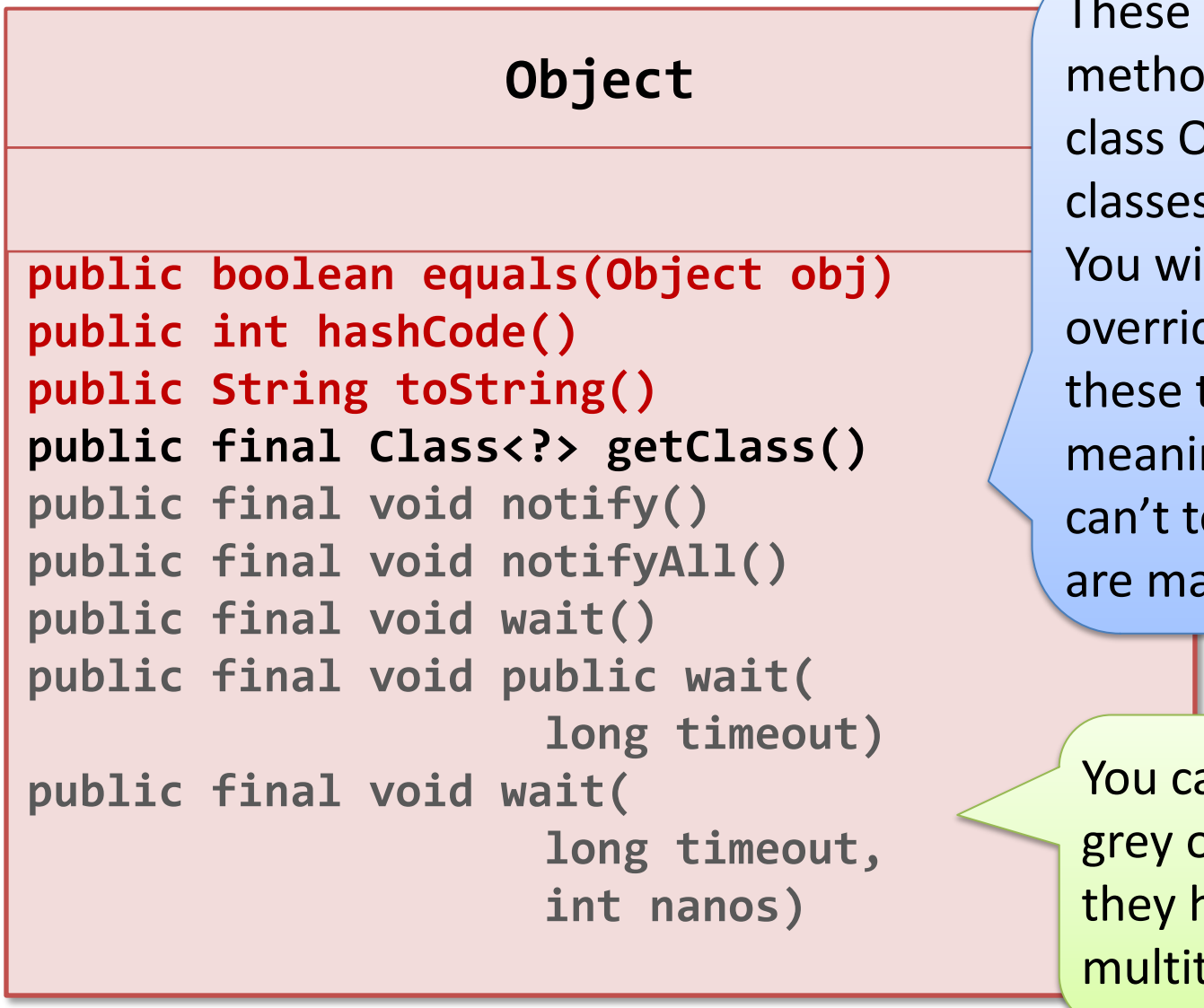
```
Object object = new Object();
System.out.println("object.getClass() returns: "
                   + object.getClass());

Cell cell = new Cell();
System.out.println("cell.getClass() re
                   + cell.getClass());
```

Here is one of the methods defined in class Object: **getClass()** which returns the full classname of the object.

```
$ java ObjectDemo
object.getClass() returns: class java.lang.Object
cell.getClass() returns: class Cell
```

Class object



These are the methods defined in class Object. All Java classes inherit these. You will have to override some of these to have real meaning. Others you can't touch: these are marked **final**

You can ignore the grey ones (for now); they have to do with multithreading

Method toString()

- You have to override toString() to give your object a String representation that suits your needs
- This: `System.out.println(myObject);`
- actually calls `myObject.toString()`
- You use it very often, especially during program development

toString() code convention

- Given this class:

```
public class Cell{
    private int size;
    private int maxSize;
}
```

- This is a typical toString() implementation:

```
public String toString(){
    return this.getClass().getSimpleName()
        + "[size=" + this.size
        + "maxSize=" + this.maxSize + "]";
}
```

Part three: enums

Enums

In a previous slide, you have seen this chunk of code creating an appropriate cell type based on the value of a String variable (provided at the command-line).

```
public void growCells(String type) {
    Cell cell;
    if (type.equals("bacterium")) {
        cell = new Bacterium();
    }
    else if (type.equals("eukaryote"))
        cell = new Eukaryote();
    }
    Cell babyCell = cell.grow();
}
```

Can you think of a scenario that would cause an error in this program?

Enums for safe predefined values

- In Java, ***enums*** have been introduced to let you specify predefined values for a given type
- For instance, if you want only bacteria, archaea and eukarya to be grown in your TestTube, you could define this as follows

```
public enum CellType{
    BACTERIUM,
    ARCHAEA,
    EUKARYA;
}
```

This defines three legal values for CellType

Creating enums

- You can't create an enum!
- For every enum value, there is ***always only one instance in the JVM!***
- You can get a reference to these instances in one of these two ways:

```
CellType bact = CellType.BACTERIUM;

CellType type = CellType.valueOf("BACTERIUM");
```

Will give an IllegalArgumentException if the type with this name is not defined

Using enum values

- You can use enums in switch/case blocks

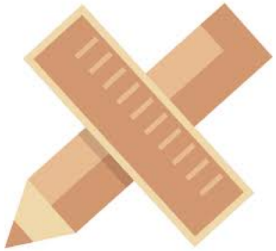
```
public void growCells( String typeStr ){
    CellType type = CellType.valueOf(typeStr);

    Cell cell = null;
    switch(type){
        case BACTERIUM: {
            cell = new Bacterium(); break; }
        case ARCHAEA: {
            cell = new Archea(); break; }
        case EUKARYA: cell = new Eukaryote(); break;}
    }
    Cell babyCell = cell.grow();
}
```

Since you "cased" all the types, you can be sure you'll have a Cell object

Enum functionality

- Besides providing single-object constants, enums can have methods to accompany each distinct value (e.g. toString).
- Have a look at the this blog for further details on enum magic: http://michielnoback.nl/the-power-of-enums/



Design rule

- Use an enum to define a static set of possible values for a given property

Summary

- This part dealt with four aspects: object construction, methods of class Object, access modifiers and enums
- Implement (a) constructor(s) when you want specific object initialization code to run and give the constructor those parameters you want client code to provide values for
- class Object defines several methods, of which toString() (and equals() and hashCode()) should be overriden in almost all classes you write