Java Programming 2
Lecture #8

Mary Ellen Foster
MaryEllen.Foster@glasgow.ac.uk
16 October 2019

Outline

Review of Objects, Classes, Inheritance

Abstract classes and methods

Final classes, methods, and fields

Exceptions

Using online resources

Objects (review)

Classes are **types;** Objects are **instances of types**

Characteristics of **objects** (real-world or software)

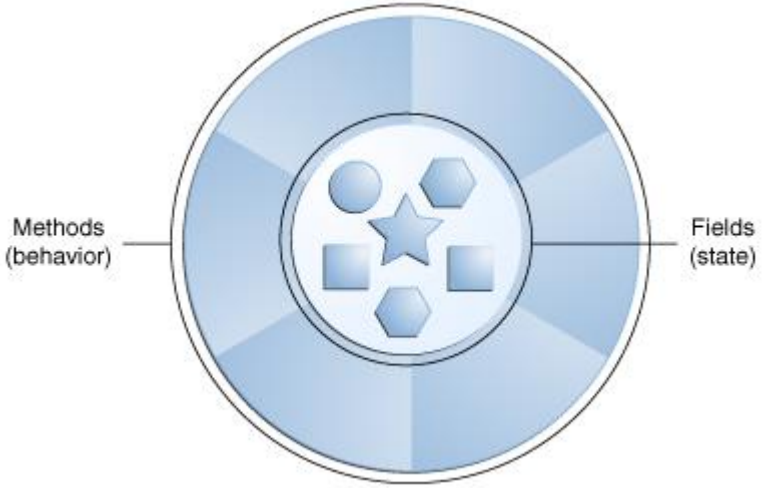
- State (fields)
- Behaviour (methods)

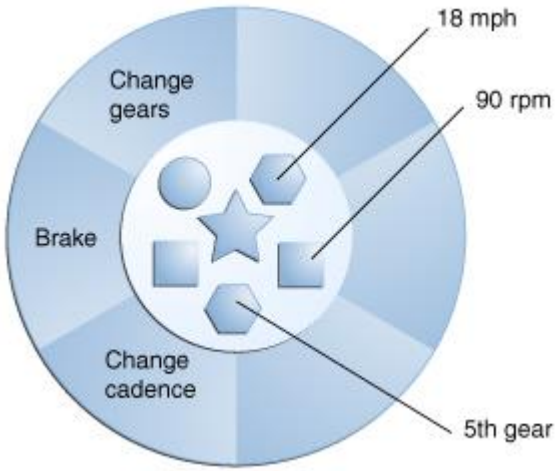
Fields:

- Store state that represent some attributes of the object
- *For `Dog` class: name, breed, size, age, …*

Methods:

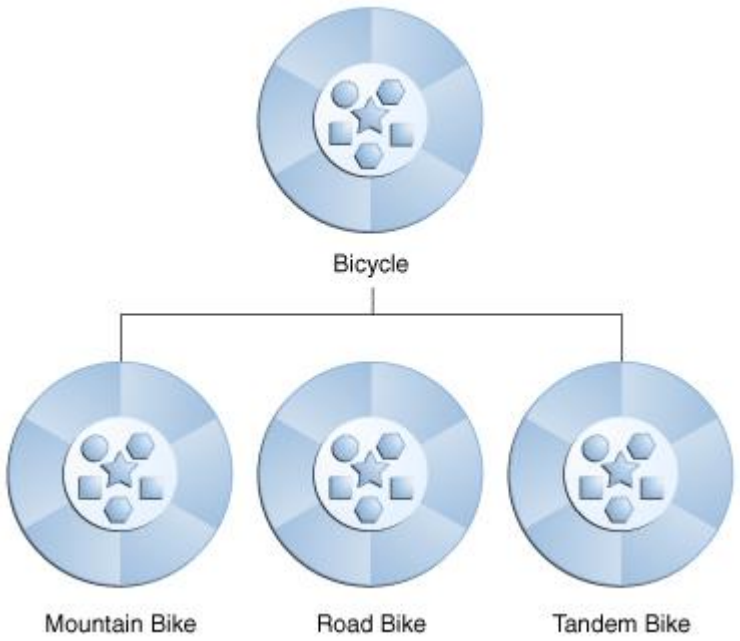
- Represent behaviour that processes and transforms the object state
- *For `Dog` class: eat(), sleep(), goForWalk(), …*





Java Tutorial "What is an Object?"

Inheritance



Java Tutorial "What is Inheritance?"

Objects (world or software) have some features in common

In OO programming, classes can **inherit** state and behaviour (fields and methods) from other classes

- Subclass is a **specialised version** of the superclass

In Java, a class can have **exactly one** superclass

- If superclass isn't specified, then it inherits from `Object`

Subclasses can **override** superclass methods to provide specialised behaviour

Constructors and inheritance

A subclass **does not inherit** the superclass's constructors

… But can call them from its own constructor using `super`

```
public class MountainBike extends Bicycle {
    public int seatHeight;

    public MountainBike(int seatHeight, int cadence, int speed, int gear) {
        super(cadence, speed, gear);
        this.seatHeight = seatHeight;
    }
}
```

If you don't add a call to the superclass constructor in the subclass, a call to the "no-args" constructor is **automatically added.**

Abstract classes and methods

Some classes have “holes” in them – methods that **must** be overridden in subclasses

Such classes are marked as `abstract`

Methods that must be overridden are marked as `abstract` too

If a subclass does not implement all `abstract` methods, it must also be marked `abstract`

First abstract watercolor, painted by Wassily Kandinsky, 1910.

Example

```
public abstract class TwoDimensionalPoint {
  protected double x;
  protected double y;

  public abstract double distanceToOrigin();
}

 public class CartesianPoint extends TwoDimensionalPoint {

   public double distanceToOrigin() {
     return Math.sqrt(x*x+y*y);
   }
 }

 public class ManhattanPoint extends TwoDimensionalPoint {

   public double distanceToOrigin() {
     return Math.abs(x) + Math.abs(y);
   }
 }
```

This method ensures that all subclasses meet a given API

- In the example, all subclasses of `TwoDimensionalPoint` must implement `distanceToOrigin()`

But: it doesn't make sense to implement `distanceToOrigin()` in the superclass

More on abstract methods/classes

Abstract methods do not have a body – just the signature followed by semicolon

```
public abstract double distanceToOrigin();
```

Abstract classes can still have

- Constructors
- Fields
- Normal (non-abstract) methods
- Static fields and methods

(Opposite of abstract)

You **cannot** create instances of abstract classes – only **concrete** subclasses

~~`TwoDimensionalPoint p= new TwoDimensionalPoint();`~~

Inheritance issues

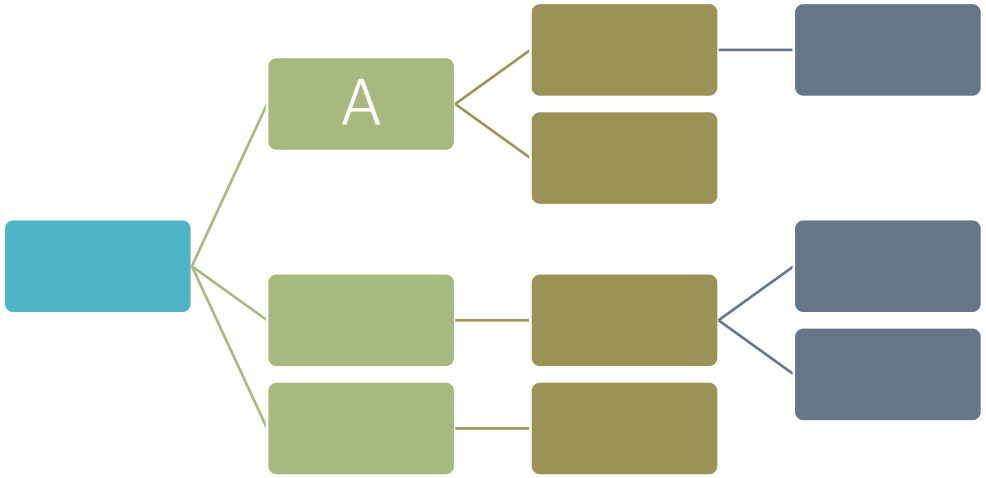
Recall: **polymorphism** means that, if code is expecting an instance of class A, you could use an instance of any subclass of A

- Subclass might override any of A's methods with its own implementation

What if A has a critical function?

- Checking passwords
- Accessing a critical piece of hardware
- …

Subclass injection attack



Example

```
public class PasswordChecker {
  public boolean check(String username, String password) {
    String passwordHash = hash(password);
    String correctHash = lookupHash(username);
    return (passwordHash.equals(correctHash));
  }
}

public class DodgyChecker extends PasswordChecker {
  public boolean check(String username, String password) {
    return true;
  }
}
```

Solution: the `final` keyword

If a method is marked as `final` then it **cannot be overridden**

- Provides predictable behaviour
- Especially relevant where method has security implications

If a class is marked as `final` then it **cannot be subclassed**

- Particularly useful for **immutable** classes such as `String` or `Double`
- … Or if all methods would require `final`

Improved password checker

```
public final class PasswordChecker {

  public boolean check(String username, String password) {

    String passwordHash = hash(password);
    String correctHash = lookupHash(username);
    return (passwordHash.equals(correctHash));

  }

}
```

or

```
public class PasswordChecker {

  public final boolean check(String username, String password) {

    String passwordHash = hash(password);
    String correctHash = lookupHash(username);
    return (passwordHash.equals(correctHash));

  }

}
```

`final` fields, parameters, and variables

If a **field** is declared `final`, then its value can never be changed

- Value can only be set at declaration time or in a constructor

If a **parameter** is declared `final`, then its value can never be changed inside the method

If a **variable** is declared `final`, then its value can never be changed

- Value can be set at declaration or later, but can never be changed thereafter

```
public class Test {
        private final int field1 = 1;
        private final int field2;

        public Test (final int arg) {
                this.field2 = arg; // okay
                this.field1 = 5;   // error

                arg = 3;           // error

                final int foo;     // okay
                final int bar = 2; // okay

                foo = 3;           // okay
                foo = 4;           // error
                bar = 4;           // error
        }
}
```

What about `static final`?

Generally used to define **constants**

- `final` modifier means that the value cannot change
- Constant names are (usually) written in `ALL_CAPS`

Examples:

- `Math.E` *The double value that is closer than any other to e, the base of the natural logarithms*
- `Long.MAX_VALUE` *A constant holding the maximum value a* `long` *can have,* $2^{63}-1$
- `System.out` *The "standard" output stream*

Summary of OO-related Java keywords

```
class
extends
public/protected/private
static
abstract
final
this
super
```

Exceptions

When an error occurs in program execution, an `Exception` is **thrown**

(Exceptions are also Java objects like any other; parent class is `java.lang.Exception`)

Unless the exception is **caught**, the entire program will crash

A sample Exception …

```
Scanner s = new Scanner (System.in);
int n = s.nextInt();
```

- 1.5
- Abc
- 11111111111111111111

```
Exception in thread "main"
java.util.InputMismatchException
at java.util.Scanner.throwFor(Unknown Source)
at java.util.Scanner.next(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at java.util.Scanner.nextInt(Unknown Source)
at Test.main(Test.java:8)
```

Details of `Scanner.nextInt()`

nextInt

`public int nextInt()`

Scans the next token of the input as an `int`.

An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Returns:

the `int` scanned from the input

Throws:

`InputMismatchException` - if the next token does not match the *Integer* regular expression, or is out of range

`NoSuchElementException` - if input is exhausted

`IllegalStateException` - if this scanner is closed

http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html#nextInt()

Java details

```
public class Scanner {
        // …
        public int nextInt()
                throws InputMismatchException,
                NoSuchElementException,
                IllegalStateException
        {
                // …
        }
}
```

Checked and unchecked exceptions

UNCHECKED EXCEPTIONS

Do not need to be explicitly handled

- Program will still compile and run without any special handling

Generally indicate **programming/logic bugs** that an application cannot reasonably recover from

Example: `ArrayIndexOutOfBoundsException`

CHECKED EXCEPTIONS

Must be explicitly handled

- Program will not compile unless you deal with them somehow

Generally indicate conditions that a well-written application should anticipate and recover from

Example: `FileNotFoundException`

More on checked/unchecked

"Unchecked Exceptions – The Controversy"
https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html

If a method specifies a checked exception, that is part of the method's public interface – anyone who calls that method should deal with exceptional cases

Why not just make everything checked?

- Runtime (unchecked) exceptions represent **programming problems**
- They can occur **anywhere** in a program and can be **numerous**
  - *e.g., in theory, every time you do anything on any object it could throw a NullPointerException*

Why not just make everything unchecked and not worry about try/catch?

- Client code should be prepared to deal with "expected" exceptional cases (file not found, device not turned on, …)

Handling exceptions #1: Catching

- Wrap a `try {}` block around any code that might throw an Exception
- Must be followed by one (or more) `catch {}` blocks
  - First one whose parameter matches the thrown exception is executed
- Optional `finally {}` block
  - Executed after entire rest of the try block

```
try {
        // code that might
        // throw Exception
} catch (Exception ex) {
        // deal with it
} finally {
        // clean up
}
```

Handling exceptions #2: Passing on

If you do something that might throw an exception, you can add that exception to the `throws` clause of the current method

Then anyone who calls your method will need to handle the exception (by catching or passing on)

```
public void doSomething()
      throws IOException
{
      // code that might
      // throw IOException
}
```

Getting the details of an Exception

Every Exception has

- A **message** (`Exception.getMessage()`)
- A **call stack** – the sequence of method calls that ultimately resulted in the error

If you use `ex.printStackTrace()` inside a handler, it will print the stack trace

- Often has line numbers, at least in your own code
- Helpful for debugging!

```
Exception in thread "main"
java.util.InputMismatchException
 at java.util.Scanner.throwFor(Unknown
Source)
 at java.util.Scanner.next(Unknown
Source)
 at java.util.Scanner.nextInt(Unknown
Source)
 at java.util.Scanner.nextInt(Unknown
Source)
 at Test.main(Test.java:8)
```

Handling exceptions: summary

OPTION #1: CATCHING

```
public void doSomething() {
        try {
                // code that might
                // throw IOException
        } catch (IOException ex) {
                ex.printStackTrace();
                // clean up
        }
}
```

OPTION #2: PASSING ON

```
public void doSomething()
        throws IOException
{
        // code that might
        // throw IOException
}
```

Throwing an Exception

Use the **`throw`** keyword:

```
throw new Exception ("Invalid input");
```

You can throw an Exception at any point in your code

String parameter indicates the message (available through `ex.getMessage()`)

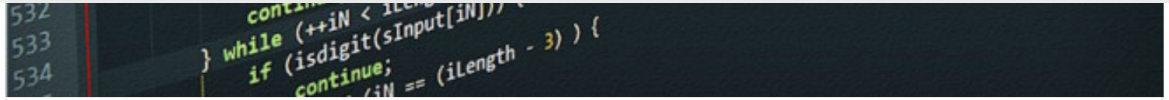
If you throw a checked Exception, you also need to add it to the header of your method with the **`throws`** keyword

```
public String processInput (String input) throws Exception { … }
```

Advantages of using Exceptions

1. Separating out error-handling code
   Instead of a series of if/then/else statements
   Just “assume” that things will work and deal with errors elsewhere
2. ***Propagating*** errors up the call stack
   i.e., sending errors along until they reach a method that is prepared to handle them
3. Grouping error types
   Exception is a class, and can be subclassed
   => different types of Exceptions can be conceptually grouped together
   (I/O exceptions, for example)

Online resources



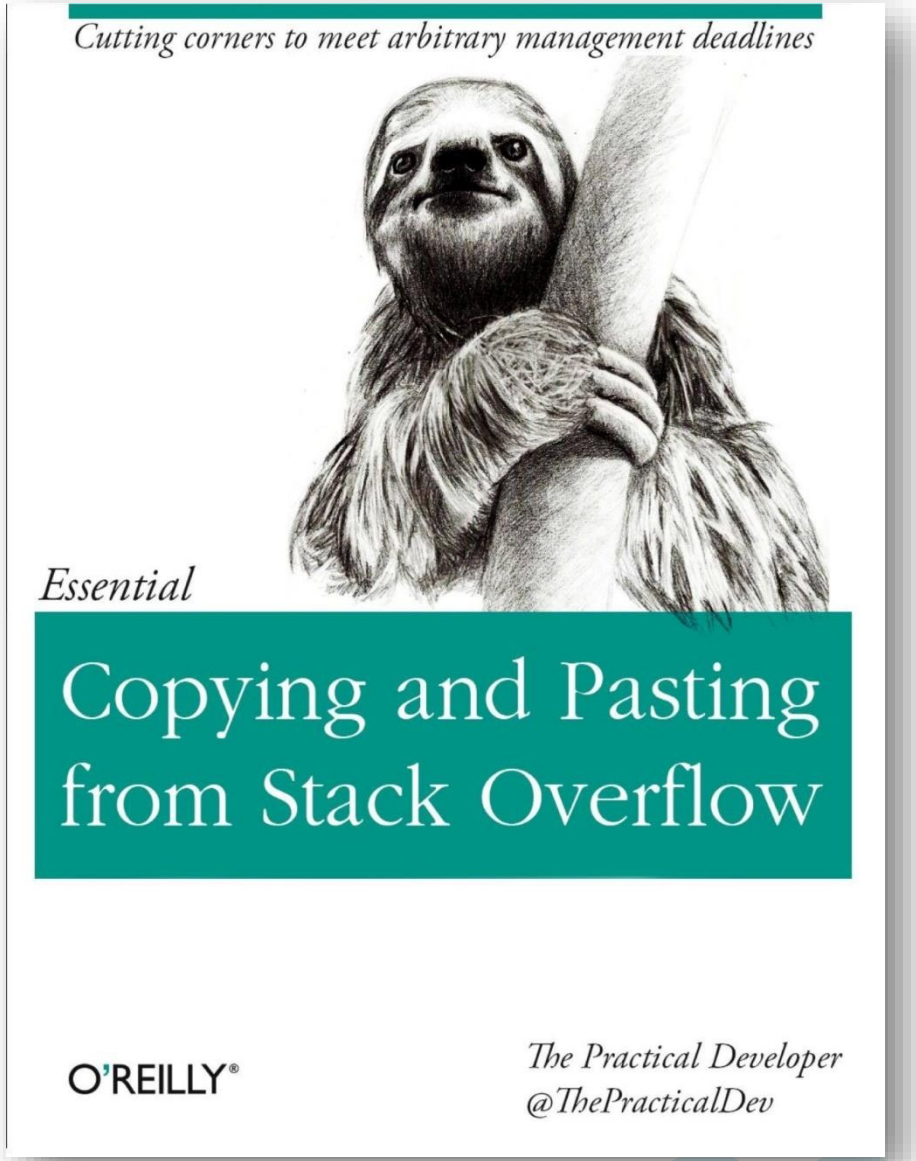
Computer Programming To Be Officially Renamed "Googling Stackoverflow"

1.9k SHARES | Facebook | Twitter 16 | Google+ 1.1k | LinkedIn 835 | Reddit 0

Washington DC – The IEEE have produced a report today where they strongly recommend that from now on, the discipline of Computer Programming should be officially renamed to "Googling Stackoverflow".

"We are recommending a root-and-branch name change to this discipline", said President of the IEEE, Thomas M. Conte. "We are even going to change the official name of the IEEE Computer Society to the IEEE Quick Look At StackOverflow Society".

"We are furthermore recommending that all universities across the planet should cease to award Bachelors degrees, Masters or PhDs in computer programming or software engineering and instead they should award these degrees in Googling Stackoverflow."



Example

Search for “java convert string to int”

First hit: http://stackoverflow.com/questions/5585779/converting-string-to-int-in-java

First response to question is:

```
int foo = Integer.parseInt("1234");
```

2556

See the Java Documentation for more information.

(If you have it in a `StringBuilder` *(or the ancient* `StringBuffer`*), you'll need to do* `Integer.parseInt(myBuilderOrBuffer.toString());` *instead).*

General rule for using online resources

Any Java techniques that you find online can in principle be used

Just make sure that …

- You **understand** what the code does – if you're using something we haven't covered yet in lectures, we won't help you get it working
- It doesn't require **any additional libraries** beyond what's provide with a standard Java development environment
- You are not **copying a full solution to the assignment** from the internet (plagiarism!)
  - *It is unlikely that online code **exactly** matches the assignment spec*
- You are not **using a library that makes the entire assignment trivial** (e.g., an online credit card checker or similar for Lab 2)
- You **acknowledge your sources** (e.g., link for answer is https://stackoverflow.com/a/5585800)

Not recommended to ask questions on StackOverflow yourself – but your question has probably been answered if you search properly

Next time

Friday tutorial: going over lab 3; discussing lab 4

Monday lecture: Packages