About the Course

Overview

The high level goal of this course is to learn how to transition from coding for courses to coding in the real world.

The Plan

- Object Oriented Programming (review)
- Advanced Java
- Tools of professional development
- Design patterns
- Clean coding practices
- Software development processes
- …and more?

What do I know about the real world?

A lot! Teaching is my hobby; writing code is what I actually do for a living. I am teaching you the things I wish I had known when I graduated.

What people say about CSC-430

> "Candidates used to struggle to get through the technical portion of interviews…now the technical questions are largely useless, because students do so well now…"

What people say about CSC-430

> "…we started probing the students to find out what had changed and the answer was CSC-430"

A local employer (paraphrased)

Grading

Component	Weight
Assignments	68%
Exams	28%

Component	Weight
Misc	4%

Textbooks

For this course, we will be using the following textbooks, in addition to free online resources:

- Head First Design Patterns, 2nd Edition, Freeman & Robinson
- Effective Java, 3rd Edition, Bloch

Resources II

You are expected to read **everything** that is assigned. If you don't:

- Do plan on failing.
- Do not plan on sympathy.

Help

Don't be afraid to ask questions!

I am available during my office hours and online throughout the day via email and the #csc430 slack channel.

Additional office hours can be planned in advance, so contact me!

Help II

Don't be afraid to ask questions!

Help III

Don't be afraid to ask questions!

Important Class Policies

- Late submission penalty: **50%**
- Plagiarism will be punished as severely as possible

Staying Up To Date

Canvas will be used to handle the general organization of the course and all **critical** announcements.

Other reminders, notes, etc. may be distributed via twitter @msupwright4 and slack.

Final Note

My goal is to push you hard. Easy classes are not worth the money you are paying and a degree with no actual skills is worthless.

Be responsible. Ask questions. Do your work.

How You Code Now

Why Are We Here?

To understand why this class exists, we need to first analyze how **you** code now, and then we can talk about why this does not scale beyond the classroom.

Code Volume

Most of your coding experience probably consists of projects with less than 100 lines of code.

How do you manage this code?

Code Volume II

Code Contribution

Code Volume III

Code Contribution

Code Volume IV

Some line counts for a few projects I work on:

- 20,185
- 269,924
- 271,335
- 173,568

Code Volume V

At these sizes, just working with the code (distributing, sharing, etc.) becomes a non trivial task.

We need a way to backup our code, track changes, and avoid conflicts with coworkers–at scale.

Testing

How do you test your course projects?

Testing II

Manually testing large codebases is, literally, not possible without doing a poor job.

The number of paths in your code to test grows exponentially! If the time you spend testing does not, then **you are not testing your code**.

Testing III

Even writing tests for large codebases is not possible if the code is written poorly.

Accordingly, we need to **automate** testing *and* write our code in a way that makes it feasible to write sufficient tests.

Building

How do you build your code?

Do you even *know* how you build your code?

Building II

As projects grow in size and complexity, even compiling, building, and deploying your code becomes a problem.

We can not rely on manual steps!

Building III

Instead, we must **automate** the build process (including testing!) to ensure that we can deliver code in a reproducible, safe way.

Ideally, we automate deployment as well.

Maintenance

How hard is it to maintain your code after a year?

You don't know, because you throw it away after a week!

Maintenance II

In the real world, your code will live for years (or decades) and will have to be maintainable by the unlucky individual that gets stuck with your legacy code.

Maintenance III

Often, **you** are that unlucky individual.

Also often, **you** will not even understand your code if you are not careful with how you write it.

Maintenance IV

Time Travel

Solution

We can largely conquer these problems (and more) by simply **caring** about our code and **automating** all the things.

Course Thesis

> Humans suck at coding and we must humbly accept all of the help we can get from tools, processes, etc.

Intro To Maven

Maven

According to its own website…

> Apache Maven is a software project management and comprehension tool [...] can manage a project's build, reporting and documentation from a central piece of information

Maven II

We will boil that down to the following, though:

> Maven is a dependency management and build tool.

Dependency Management

What is *dependency management*?

Dependency Management II

Code you work on for your courses is often completely self contained, in one or two class files.

You will typically only be importing other classes from the standard library.

Dependency Management III

In a real world project, though, you will typically be relying on a *significant* amount of code written by others.

Dependency Management IV

This code will be packaged in *jar* files which you will need to have available when building and distributing your code.

Dependency Management V

In the bad old days, this meant:

- You had to find the libraries
- You had to download them
- You had to keep track of them
- You had to ensure *their* dependencies are included
- You have to make sure to include them properly when compiling you code

Dependency Management VI

This may not sound to bad, but on large scale projects, this can be a huge source of problems!

Dependency Management VII

A *dependency manager* will allow you to provide a small amount of configuration, and it will then handle all of these problems for you *in an automated manner*.

Build Tools

When we talk about a build tool, we are generically referring to any tool that allows you to provide a *configuration* (or script), which can then handle all build steps that are necessary to produce your end product.

Build Tools II

For instance, you could use a build tool to trigger dependency management, compile your code, execute automated tests, package your compiled code, and more!

Automation

A keep theme here is *automation*.

If our build process is too complicated, we *will* forget steps and make mistakes.

This will lead to inconsistencies and errors.

Automation II

Complex manual processes also make it difficult to work with collaborators, because it takes significant work just to get the code running the same on all developer machines.

Automation III

Instead, we use a clear, precise configuration and feed it to a build tool to guarantee that we have *reproducibility* anywhere our code is built.

This also allows us to reduce our build process to a single command!

Maven III

There are usually multiple build tools that can be used for any given programming language, but Maven is one of the most commonly used in the Java world.

You may be interested in becoming familiar with *Gradle* as well, though.

Project Object Model

Maven relies on a configuration called a *Project Object Model (POM)* file.

Our main concern at this point is how to configure dependencies.

For simple projects, the building works out of the box!

Coordinates

To add a dependency, we simply need to provide the *group id*, *artifact id* and *version* of the library you want to use.

We call these the *coordinates* of the artifact.

Coordinates II

For example, we might add a dependency on a course library like:

```
<dependencies>
  <dependency>
    <groupId>edu.murraystate</groupId>
    <artifactId>BlobAPI</artifactId>
    <version>1.0</artifactId>
```

```
  </dependency>
</dependencies>
```

Transitivity

Note that, when you add a dependency, it may also need its own dependencies.

Fortunately, Maven artifacts are packaged with their own POM file, so Maven will go ahead and download *all* dependencies *transitively*.

Repositories

Maven is configured, by default, to pull artifacts from *Maven Central*, which is a public, centralized artifact repository.

You may, however, need to use custom, private repositories.

Repositories II

```
<repositories>
  <repository>
    <id>BlobAPI-mvn-repo</id>
    <url>https://raw.github.com/MSUCSIS/csc430-maven/mvn-repo/</url>
    <snapshots>
      <enabled>true</enabled>
      <updatePolicy>always</updatePolicy>
    </snapshots>
  </repository>
</repositories>
```

Design Patterns

Design Patterns

Design patterns are, ultimately, nothing more than the result of applying a few object oriented principles which you should try to follow.

Design Patterns II

We will be learning those principles, but it is also good to study some of the patterns that arise from their use so that we don't have to reinvent the wheel.

Design Patterns III

Learning established patterns allow you to quickly get to a solution, and they also provide you and other developers with a *shared vocabulary* that can be used to discuss your code.

Design Patterns Are Garbage?

During the semester, we will also be discussing how design patterns are actually kind of awful and may be seen as a "least awful" solution to problems in many cases.

Strategy Pattern

> The *strategy pattern* defines a family of algorithms, encapsulates each, and makes them interchangeable. A strategy lets the algorithm vary independent from clients that use it

Strategy Pattern II

Before we take that definition apart, let's take a look at the object oriented principles that lead to this pattern:

- Encapsulate what varies
- Program to interfaces
- Favor composition over inheritance

Encapsulate What Varies

Modifying code is usually pretty dangerous, because it can introduce regressions in your code.

Encapsulate What Varies II

If we do not properly encapsulate our code, then simple changes can have far reaching impact.

Accordingly, we would prefer to identify behavior that may vary in our code and isolate it from code that does not vary.

Encapsulate What Varies III

If we do this successfully, then the non varying code can be written, tested, and left alone forever.

...assuming our testing was sufficient

Example

For instance, let's assume we have the following code

```
public class Duck{
  private final int id;

  public Duck(final int id{
    this.id = id;
  }

  public String fly(){
    // Let's imagine this is a more complex computation
    return "I'm flying";
  }

  public int getId(){ return id; }
}
```

Example II

In this example, we can be relatively sure that the id related code will not change. However, it is not unlikely that different ducks might fly differently.

Example III

So, we might do this instead

```
public class Duck{
  private final int id;
  private final FlyBehavior flyer = new FlyBehavior();

  public Duck(final int id){
    this.id = id;
  }

  public String fly(){ return flyer.fly(); }
```

```
  public int getId(){ return id; }
}
```

Example IV

```
public class FlyBehavior{
  public String fly(){
    return "I'm flying";
  }
}
```

Now, if the flying behavior needs to change, we still never have to handle the Duck class again (almost…).

Example V

We still have a slight problem. Our Duck can only store the class FlyBehavior, so the result isn't *that* flexible.

How can we support different flying behaviors?

Program to an Interface

If we write code which uses specific, concrete types, we are stuck with those types and have to manually modify our code, duplicate code and do other awful things to use other types.

Program to an Interface II

If, instead, we program to more abstract interfaces, we can modify the *behavior* of our code, without modifying the code itself.

Program to an Interface III

Note that, in this context, *interface* refers to a conceptual interface which can be coded in the form of

- an interface
- an abstract class
- a common super type
- a function signature

Example VI

```
public class Duck{
  private final int id;
  private final FlyBehavior flyer;

  public Duck(final int id, final FlyBehavior flyer){
    this.id = id;
    this.flyer = flyer;
  }

  public String fly(){ return flyer.fly(); }
  public int getId(){ return id; }
}
```

Example VII

Now, we can create various subclasses for *FlyBehavior* and swap them in and out to customize how ducks fly.

Even cooler, we can do this *at runtime*!

Favor Composition

When we want to extend the behavior of our code, we typically have two ways to do it. We can either use *inheritance* or *composition*.

Inheritance

Using inheritance, A class A can extend another class B and override or add methods to obtain the desired behavior.

We often say that such an instance of A "is a" B.

Composition

We could instead *compose* objects and include a field of type C inside of a class B and the use that to change the behavior of the class B.

In this case, we would say that B "has a" C

Favor Composition II

We will learn that it is often valuable to write software so that different components are "decoupled" from each other. This is almost always easier when using composition.

Strategy Pattern III

At this point, we have reached a nice clean solution which is, in name, the *strategy pattern*.

As you see, we were able to reach this point only using basic principles, but it would have been easier to just jump straight to this design!

When To Use

You should consider using the strategy pattern when:

- Many classes differ only in some type of behavior
- You need different variations of an algorithm
- An algorithm uses data that clients shouldn't know about
- A class defines many behaviors selected by a conditional

Problems

Some drawbacks of the strategy pattern include:

- clients must be familiar with the strategies
- various strategies may require different parameters
- If stateful, there could be a large number of classes required

Is It Garbage?

We know that an interface that contains a single method can be replaced with a lambda expression.

This means we don't *have* to define a special interface for the strategy: we just need to specify a general function interface as a parameter.

Is It Garbage? II

Additionally, we are passing strategies into constructors, but we could simply pass them into the methods where they are needed.

Storing them in a field is just a convenience.

Is It Garbage? III

At this point, we have essentially reached the basic concept of *higher ordered functions*

> A higher ordered function is a function which takes another function as a parameter

Is It Garbage? IV

If the strategy pattern is basically just a complicated implementation of higher ordered functions, then what is the point of jumping through extra hoops?

As Java incorporates more functional concepts, design patterns like this start to become much less interesting.

Java Things

String Representation

Java provides a default *toString()* method for all *Object*s, but it leaves much to be desired.

So, it's usually a good idea to override the *toString* method to give a better *String* representation.

String Representation II

This is particularly useful when debugging your code, as you can quickly get a summary of the state of an *Object* without digging into the debugger data.

Building Strings

So, how should you build complex Strings?

For simple cases, it might be fine to do

```
final String s = "Point(" + x + "," + y + ")";
```

But it's mainly fine because the compiler can optimize it. Why might this be bad?

Building Strings II

To make things a bit cleaner, though, you may want to use *String.format*:

```
final String template = "Point(%d,%d)";
final String s = String.format(template,x,y);
```

Building Strings III

For building more complex strings that involve concatenation and iteration, you're probably better off using a *StringBuilder*:

```
final StringBuilder sb = new StringBuilder();
sb.append("Points:");
for(final Point p : points){
  sb.append(p.toString());
  sb.append(",");
}
sb.setLength(sb.length()-1);
final String s = sb.toString();
```

Review

What is the difference between calling

`p1.equals(p2)`

and

`p1==p2`

What Does Equality Mean?

When we talk about values being equal, we are usually making a lot of assumptions about what that means.

In reality, there is no good, universal definition of equality.

Equality

For instance, all of the following could meet the definition of two points being equal:

- They are stored at the same memory location
- They have the same x and y components
- They represent the same point in space

Which one is correct?

Equality II

Java, by default, assumes that *equals* means two items are *stored at the same memory location*. So, in the following code:

```
final Point p1 = new SimplePoint(10,10);
final Point p2 = new SimplePoint(10,10);
```

The objects *p1* and *p2* are *not* equal!

Structural Equality

Intuitively, this may seem odd. Accordingly, Java allows you to override the *equals* method and make it work however you want.

Often, the best decision will be to use *structural equality*: two objects are equivalent if they have the same structure and values.

Equals

For example, we could write the following method:

```
public boolean equals(final Point other){
  if(getX() == other.getX() && getY() == other.getY()){
    return true;
  }else{
    return false;
  }
}
```

Equals II

So, is our equals method good enough?

Equals III

The *equals* method is actually defined with a parameter of type *Object*, so we can't expect to receive a *Point*!

Also, we have not checked for the possibility of receiving a *null* value!

Equals IV

```
public  boolean equals(final Object other){
  if(!(other instanceof SimplePoint)){
    return false;
  }else{
    final SimplePoint p = (SimplePoint)other;
    // Why ==?
    return x==p.x && y==p.y;
  }
}
```

Equals V

We can also short circuit the check by adding the following:

```
if(this==other){
  return true;
}
```

Semantic Equivalence

We could instead attempt to use some sort of semantic equivalence, but it is often dangerous to do so and may not return the intuitive result for your users. For example,

```
final Currency brl = new BRL(1.0);
final Currency usd = new USD(0.247114);
// Should this be true due to the current fx rate?
brl.equals(usd);
```

Equals In General

Generally speaking, however we choose to define *equals*, it should be *reflexive*, *symmetric* and *transitive*.

Which you, of course, remember from CSC-300, right?

Reflexive

$$\forall x : x \equiv x$$

In other words, an object must equal itself.

Symmetric

$$\forall a, b : a \equiv b \iff b \equiv a$$

In other words, objects must agree on being equal.

Transitive

$$\forall a, b, c : (a \equiv b \land b \equiv c) \Rightarrow a \equiv c$$

In other words, we can infer equivalence via other equivalences which have an "overlapping" element.

Inheritance

When we use inheritance and add fields to an object, it almost guarantees that we have broken the *equals* method if we expect it to work with all the parent and child classes.

Immutable Objects

Immutable objects also work really nicely with structural equality, because you can literally swap equal instances with no impact on your code.

(assuming you don't use ==, or reflection, or something else nasty)

Immutable Objects II

It also means that objects can be re-used. We could, for example, use the exact same *Point(0,0)* object for every instance of a *Duck* at that location over time.

Immutable Objects III

If we have a small, finite, number of possibilities, we can just create a pool of *Point*s and never have to allocate a new one!

Immutable Objects IV

This explains the curious *Integer.valueOf(int i)* method. This:

```
final Integer x = new Integer(10);
```

Allocates a new *Integer* on every call. On the other hand:

```
final Integer x = Integer.valueOf(10);
```

Can cache common values and reuse them.

Don't Forget Hash Code

The *hashCode* and *equals* methods are expected to work in concert:

If two objects are "equal", then they must return the same hash code!

(This does not mean they must have different hash codes if they are unequal!)

Hash Code

With structural equality, this is not difficult: you just need to pass all fields used for equality testing to *Objects.hash(…)* and it will compute a good hash code for you.

Hash Code II

With more exotic definitions of equality, this becomes very difficult.

Also, with immutable objects, we only really need to compute the code once!

Gotchas

There are a lot of nice ways to shoot yourself in the foot when using Java. Let's take a look at some reasonable looking code and figure out why it may not be so reasonable.

Equals

```
public boolean test(final String x){
  return x=="something"
}

// ...
```

```
System.out.println(test(x));
```

What output will we see if x has the value "something"?

Equals II

Depending on the code and how the compiler optimizes, we may get true or false. If x gets its value from a literal and the compiler optimizes the code so that duplicate *String* literals are stored at the same location, we'll get true. Otherwise, false.

Casting Arrays

How's this code?

```
final Object[] objects = {"hello","ola","Hallo"};
final String[] strings = (String[])objects;
```

Casting Arrays II

It's awful and it's guaranteed to throw an exception!

```
final String[] strings =
  Arrays.asList(objects).toArray(new String[]);
```

Threading

```
public class X extends Thread{
  private boolean go = true;
  public void run(){
    while(go){
      // do something
    }
  }

  public void dontgo(){
    go = false;
  }
}
```

Threading II

```
public static void main(final String[] args){
  final Thread t = new X();
  t.start();
  t.dontgo();
}
```

Threading III

This code may never stop, because the memory modification from the main thread is not guaranteed to be observed by all other threads (including the actual thread itself!)

It's only guaranteed to work if we make the variable *volatile* or insert synchronization code.

More Threading

```
public class Y{
  private int n;
  public Y(final int n){
    this.n=n;
  }

  public void test(){
    if(n!=n){
      throw new AssertionError();
    }
  }
}
```

More Threading II

Assuming we are sharing this object with other threads, what will happen when we call *test*?

Are you sure?

More Threading III

In fact, this may throw an assertion error, because there is no guarantee that both accesses to n will see the same value, even though only one value was

assigned! Why?

LOL

```
public class Lol{
  public double value;
}

final Lol = new Lol();
Lol.value = 12121.0;
```

LOL II

Assuming this object is shared with other threads, what possible values might they see?

LOL III

They could see 0.0, which is the default for a double.

They could see 12121.0, which is what it's updated to.

...or they could see whatever you get from taking half the bits from 0.0 and half the bits from 12121.0

** LOLOLOLOLOLOL **

Immutable Objects II

Of course, immutable objects using *final* variables avoids all of these problems!

Takeaway

Programming is awful and there's no way you can write correct code without a lot of hard work and help from tools, best practices, etc.

Also

Buy Java Concurrency in Practice before you try to write any multi-threaded Java code in the real world.

If you aren't using Java, make sure you look into the "memory model" of your language: it's almost guaranteed it's a horror show as well!

Observer Pattern

Observer Pattern

> defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

AKA: Publish-Subscribe, Dependents

OO Principle: Loose Coupling

> Strive for loosely coupled designs between objects that interact.

Loose Coupling

- Allows objects to interact with minimal knowledge
- Changes to one component won't impact another (within reason)
- Components can be added without requiring changes to code
- *Turns dependencies into a runtime property instead of a static property*

OO Principles

We will also be utilizing the previous principles we learned:

- Encapsulate what varies
- Favor composition over inheritance
- Program to interfaces

Strategy Pattern

Let's think about how the strategy pattern works real quick, and see if we can derive the observer pattern from it.

Strategy Pattern II

Whenever some action happens (e.g., a method call), we delegate to a strategy which encapsulates the desired behavior.

Observer Pattern II

Whenever some action happens (e.g., *an update*), we *notify* an *observer* which encapsulates the *dependent* behavior.

Sounds familiar…

Observer Pattern III

As long as we only have one dependency, we can see that the observer pattern and strategy pattern are basically identical.

One-To-Many

So, how do we incorporate *multiple* dependencies into our strategy pattern?

We just add the ability to provide multiple strategy patterns to a subject!

One-To-Many II

```
public class Duck{
  private final long id;
  private final Observer observer1;
  private final Observer observer2;

  public Duck(long id,Observer o1,Observer o2,Observer o3){
    // ...
  }
  public void fly(){
    observer1.notifyOfFlight(id);
    observer2.notifyOfFlight(id);
  }
}
```

One-To-Many III

We will also typically provide the ability to *subscribe* and *unsubscribe* observers at run times, though this is not particularly different from the strategy pattern (though we often make strategy patterns final).

One-To-Many IV

```
public class Duck{
  private final long id;
  private final List<Observer> observers=new ArrayList<>();

  public Duck(long id){
    this.id = id;
  }

  //...
}
```

One-To-Many V

```
public class Duck{
  //...
  public void fly(){
    for(final Observer o : observers){
      o.notifyOfFlight(id);
    }
  }
}
```

So just use strategies?

No! While the essence of these two patterns are the same, the *use cases* for the observer pattern often have common requirements (like (un)subscribing) and specific semantics related to when notifications should occur, etc.

Strategy vs. Observer

Observers...

- Almost always have a greater number of dependents
- Have a looser coupling, because the intent of the dependent is unknown
- Focus more on the *dependency* and less on the *computation*

I.e., a subject does not care what the dependency is doing!

When To Use

You should consider using the observer pattern when:

- An abstraction has two aspects where one is dependent on the other.
- When a change in one components requires changes in others, but which/how many is not known.
- When objects should be able to send/receive data without assumptions.

Problems

One issue with the observer pattern is that we really have no idea what dependents are doing. If an observer performs a large amount of computation, or cascades updates to its own observers, you can cause a huge performance hit!

Implementation Details

We also must decide whether we want to use a *pull* model or a *push* model for our updates.

Pull

- Observers only receive an update notification and reference to the subject
- Observers may then query the subject for just the data they need
- Embraces the subjects ignorance of its dependents' behaviors
- Could be less efficient for the observer, due to the overhead of gathering data

Push

- All of the needed data is pushed to all observers
- Makes the coupling a bit tighter, perhaps
- Gives the subject more control over the data

Decorator Pattern

The Decorator Pattern

> Attaches additional responsibilities to an object *dynamically*. Decorators provide a flexible alternative to subclassing for extending functionality.

AKA: Wrapper

OO Principle: Open/Closed

Classes should be open for extension, but closed for modification

The Decorator Pattern II

- Allows for extension at *runtime*.
- Allows you to modify the behavior of a class without actually changing it
  - …and without it even being aware of the decorators existence!

Example

```
public interface Duck{
  String fly();
}
```

Example II

```
public class BasicDuck implements Duck{
  private final FlyStrategy strategy;
  public BasicDuck(final FlyStrategy strategy){
    this.strategy = strategy;
  }

  // ...
}
```

Example III

```
public class StrongDuck implements Duck{
  private final Duck wrapped;
  public StrongDuck(final Duck d){
    this.wrapped = d;
  }

  public String fly(){
    return wrapped.fly() + wrapped.fly();
  }
}
```

The Decorator Pattern III

- Avoids an explosion of subclasses
  - $ |base classes| \times |variations| $ (if we only use one variation)
  - $|baseclasses| \times 2^{|variations|}$ (any combination of variations)
  - ∞ (if we allow repeated variations)

The Decorator Pattern IV

- Allows for added behavior before/after the components behavior
- Allows for the complete replacement of a component's behavior
- Allows for enhancement of a given *instance* of a class.
- Completely transparent to clients (unless they've done something ugly)

Transparency

- A decorator and the object it decorates are *unique*
- They, naturally, are *not* the same class
- Accordingly, things like referential equality and casting may fail!

When To Use

- Behavior needs to be added to an object instance *dynamically* and *transparently*
- Behaviors may need to be removed from an instance
- The number of variations make subclassing impractical (or impossible)
- The class definition is not available for subclassing.

Strategy Vs. Decorator

> We can think of a *decorator* as a *skin* over an object that changes its behavior. [...] The *strategy* pattern is a good example of a pattern changing the *guts*.

Strategy Vs. Decorator

- To use a *strategy*, the client must have knowledge of the strategy interface.
- When using a *decorator*, the client can be completely ignorant of its existence.

Factory Pattern

OO Principle: Dependency Inversion

> Depend upon abstractions. Do not depend on concrete classes.

Dependency Inversion

- Both **high level** and **low level** classes should depend on abstractions.
- Variables should not reference concrete classes
  - Which means we can't use **new**!
- Classes should not derive from concrete classes
- Methods should not override base class implementations

...for some definition of "should"

DI and Factories

With past patterns, we have been abstracting and encapsulating behavior, dependencies, etc. and making them runtime properties, increasing extensibility, etc.

To eliminate coupling introduced by the **new** operator, we will now abstract and encapsulate **object instantiation** using factories.

The Factory Method Pattern

> Defines an interface for creating an object, but let's subclasses decide which class to instantiate. The **Factory Method Pattern** lets a class defer instantiation to a subclass.

AKA: Virtual Constructor

Example

```
public interface Product{
  // ...
}

public class Car implements Product{
  // ...
}
```

```
public class Boat implements Product{
  // ...
}
```

Example II

```
public abstract class Producer{
  public Product produce(){
    Product p = createProduct();
    // do stuff with product
    return p;
  }

  protected abstract Product createProduct();
}
```

Note that the **Producer** only has dependencies on the **Product** interface, just like the **Car** and **Boat** classes. This class can now be *closed for modification* since we have inverted our dependencies.

Example III

```
public CarProducer extends Producer{
  protected Product createProduct(){
    return new Car();
  }
}

public BoatProducer extends Producer{
  public Product createProduct(){
    return new Boat();
  }
}
```

Factory Method

- Eliminates the need to bind application specific classes in code
- Allows us to close our classes to modification
- Concentrates instantiation into fewer locations

Isolating Problems

A common goal of good design is to simply take the bad stuff you can't eliminate from your code entirely, and isolate it in the smallest number of locations possible. This applies to:

- object instantiation
- side effects
- **instanceof** testing
- …

When to use

- When a class can't anticipate the class of objects it must create
- When a class wants to delegate object instantiation to subclasses

Warning

If the **only** thing the subclass does is instantiate an object, then maybe we're just making our lives more difficuly for no reason.

If you are already subclassing, though, then it's fine.

Abstract Factory Pattern

> Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Abstract Factory Pattern II

- Isolates concrete classes
- Makes it easy to exchange **families** of classes
- Promotes consistency

When To Use

- A system should be independent of how its products are created
- A system should be configured with one of multiple families of products
- A family of products **must** be used together
- You want to provide a library of products without exposing their implementation

Warning

If we need to add a new type of product, it can be difficult, because all product families must be updated.

Adapter Pattern

Adapters

We briefly discussed adapters earlier in the semester when discussing decorators, but didn't go into much detail, because there is very little difference. For the sake of completeness, let's review briefly.

The Adapter Pattern

> Converts the interface of a class to another interface expected by clients. Lets classes work together that couldn't otherwise because of incompatible interfaces

AKA: Wrapper

Adapters II

As mentioned before, the main difference between **decorators** and **adapters** is that **decorators** implement the *same* interface as what is being wrapped, while **adapters** use a different interface.

Adapters III

Note that when using **adapters** we lose the benefits of *transparency* that we obtain from using **decorators**.

Accordingly, we can "stack" **decorators**, but can only have one **adapter** of a given kind.

The Facade Pattern

> Provides a unified interface to a set of interfaces in a subsystem. Defines a higher level interface that makes the subsystems easier to use.

Facades

We can view facades as, basically, adapters which adapt multiple complex interfaces to a single, simple interface. In other words, they will often "wrap" more than one object, and expose their functionality in an adapted interface.

Testing

Testing

No matter how good you are, you **will** make mistakes. Probably more mistakes than you will ever know.

Testing II

Accordingly, it is necessary to not only do what you can to *not* make mistakes, but also to rigorously test your code to confirm that you were successful.

Testing III

You will **still** have bugs, though…

Unit Tests

Our first line of defense will be **unit tests**: small, directed tests aimed to check whether or not the smallest units of our code are working correctly.

Unit Tests II

Ideally, our development process would be *driven* by our unit tests.

This concept is referred to as **Test Driven Development (TDD)**.

Test Driven Development

- First Law: Don't write production code until you have failing tests
- Second Law: You may not write more test code than is necessary to create a failure
- Third Law: You may not write more code than is necessary to fix the failures.

Test Driven Development II

Alternatively, we can restate these laws as the following list of steps:

1. Write a test that fails
2. Fix the failure with minimal changes
3. Refactor and repeat

Test Driven Development III

If we do this well, then we know that all of the desired functionality has been tested, because, otherwise, it would not exist.

Also, we now have a safety net to protect us from **regressions** in the code base. If someone breaks our code, we should immediately know because of failing tests.

Test Driven Development IV

Unfortunately, it doesn't always go so smoothly. For instance, we have to write our tests correctly, we have to test for *unintended* results as well as intended results (which is more difficult to do!), and, in some cases, we **can't** write unit tests!

Swiss Cheese Model

In many fields, safety relies on the "Swiss Cheese" model:

There will always be holes in any layer of safety that you implement, but if you have enough layers in places, it greatly decreases the odds that a hole will exist in the entire system.

Swiss Cheese Model II

So, we *try* to use our brains to make sure we right good code, and we try to write good tests. We will fail at both, but will hopefully come closer to eliminating bugs than if we didn't test at all.

Static Typing

Another simple layer to add, which a lot of people overlook, is using the type system to model your problems.

Static Typing II

Strong Static Typing can often give you similar benefits as unit testing.

- The more you model your problem with types, the fewer invalid states will compile
- …and the easier it is to refactor code with confidence

Static Typing III

In languages like Agda, Idris, Coq, you can model problems so completely that compilation "guarantees" correctness (sort of…), because invalid states can not be compiled.

In these languages, you are basically writing proofs.

Swiss Cheese Model III

This gives us:

- Our coding skills
- The type system/compiler
- Unit tests

To combat bugs in our code

Unit Tests III

So, our unit tests will allow us to ensure code performs as expected, to modify code without causing regressions, and empowers us to refactor our and improve our code without fear.

But **how** do we write unit tests…

Unit Tests Are Code

Obviously. The point, though, is that you should use all of the principles you apply to writing good code in general when writing unit tests. You will need to maintain, fix, and improve tests over time just like all other code!

Unit Tests Are Code II

If you don't follow best practices, updating tests will become such a nightmare that you will simply start deleting tests or just stop testing at all. Then regressions creep in and the game is over.

F.I.R.S.T

- **Fast**: or they won't be executed
- **Independent**: isolate tests so it's clear what causes failures
- **Repeatable**: it should be trivial to test again and again
- **Self-Validating**: test should be reduced to pass/fail without need for human validation
- **Timely**: should be written before/along with code being tested

What To Test

First, we need to identify what a given unit should do.

This sounds simple, but it isn't!

We can easily create tests for the common, obvious cases, but we must also test for the **boundary** cases.

Boundaries

For instance, if we are implementing a function to calculate the area of a rectangle given its height and width, what inputs should we test?

First, we would do something obvious like *area(10,5)=50*.

Boundaries II

But what happens if we are given negative values? What if we are given zeros?

We often forget about these types of inputs when writing code and just assume we will "obvious" inputs.

Boundaries III

In this particular case, the logic is simple enough that we can easily address the problem with a couple of tests and some branching in our code.

In general, though, we really need to focus on our domains.

Boundaries IV

In general, for an integer parameter, we should probably consider the following values:

- positive values
- negative values
- zeros
- one
- negative one

Boundaries V

If we know we are using modular arithmetic, then we may want to test one representative from each congruence class as well.

If we are working with *mod m*, then maybe a good set of tests would be

-m-1, -m, …, -2, -1, 0, 1, 2, …, m, m+1

Boundaries VI

Basically, what you want to do is think about the input types, identify boundary values that are likely to cause different behavior, then test those boundary values, near those boundary values, and far away from the boundaries in all directions.

Boundaries VII

And don't forget *null* values!

Boundaries VIII

So what would good inputs be for a method with a *List* input?

What about a method that takes a *Point* as input?

What about *String* inputs?

Assertions

Once we have decided what we need to test, we will write a single unit test for each interesting input and make an **assertion** about what its output should be.

Assertions II

Ideally, you should have one assertion per test. if not, all assertions should at least be testing one aspect of your code.

Another Problem

How can we test code that requires a database lookup?

Problem II

As we stated previously, we want to isolate the code being tested from the rest of the code base.

A database lookup, though, will require database libraries, data model code, a running database, proper setup of the data in the database, etc.

Isolate At The Boundary

The first step should be to isolate the database code from your code. You can do this by providing some interface that matches your data model and provides convenience methods for lookups.

Isolate At The Boundary II

All SQL, driver configuration, etc. will be hidden behind this interface, so your code that requires a lookup may simply call something like:

```
final User u = db.lookupUserById(10);
```

Isolate At The Boundary III

The *db* object can then be passed into our object or method and, since we have "programmed to an interface," we can replace this object with *any instance of the same interface*.

Mocking

To properly isolate the our code from the database lookup when testing, we could provide a **mock** for the database lookup object.

In other words, we can create a class that matches the necessary interface, but does not actually interact with a database!

Mocking II

In our unit test, we will create a mock that is configured to return a *specific instance* of the *User* class when the lookup method is called with the value *10*, and then we can write our test with the assumption that the lookup succeeds!

Mocking III

If we mock all dependencies in a method or class using this, approach, then we know that any errors are caused solely by the method or class logic itself, and not by dependencies.

Mocking IV

This also makes it easier to write tests, because a lot of code my be required to create *real instances* that you could use, but *mocks* are usually rather trivial to create.

Review: Object Oriented Programming

Let's Review

For this course, being comfortable with object oriented programming concepts and having a good understand of **why** they are beneficial is required.

Interfaces

Let's start with interfaces!

First, what are **interfaces**?

Interfaces II

We could say that interfaces are "contracts" between the implementers of some code and the users of that code.

Interfaces III

In other words, in an interface, you are stating what methods you guarantee will be provided by any class implementing that interface.

Interfaces IV

Next, how do **interfaces** differ from **classes**?

Interfaces V

The old school explanation would be that interfaces can not contain **implementations** of methods. They may only contain **method signatures**.

(This isn't actually true anymore, though)

Example

```
public interface Transform {
  String apply(final String input);
}
```

Interfaces VI

This interface represents the **abstract concept** of code which performs *String* transformations.

Note that this code does not *do* anything! It just states that **any class** implementing this interface **will** provide a method called **apply** that will perform a String transformation.

Classes

Now how about classes?

We mentioned above that classes contain actual **implementation** details. In other words, they contain code that actually does stuff.

Example II

```
public class Reverse {
  public String transform(final String input){
    final StringBuilder sb = new StringBuilder();
    for(int i=input.length-1; i>=0; i--){
      sb.append(input.charAt(i));
    }
    return sb.toString();
  }
}
```

Classes II

Is this a class?

Does it perform a *String* transformation?

Does it implement the *Transform* interface?

Example III

```
public class Reverse implements Transform {
  public String apply(final String input){
    // ...
  }
}
```

Now we have stated that we are agreeing to the "contract" defined by *Transform*

Example IV

```
final Reverse transformer = new Reverse();
System.out.println(transformer.apply("Hello");
```

```
final Transform transformer = new Reverse();
System.out.println(transformer.apply("Hello"));
```

What's the difference here? Why would we do this?

Polymorphism

At this point, there is no real reason to use interfaces at all. What happens, though, when we decide to add *another* type of transformation to our code.

Example V

```
if(doReverse){
  final Reverse reverse = new Reverse();
  return reverse.apply("Hello");
}else if(doNoVowels){
  final NoVowels noVowels = new NoVowels();
  return noVowels.apply("Hello");
}else if ...
```

Polymorphism II

While all of these classes do different things, they all, at a high level, are transforming *String*s.

So, if we use the *Transform* interface for all of them, then we can unify them with one variable.

Example VI

```
final Transform transform;
if(doReverse){
  transform = new Reverse();
}else if(doNoVowels){
  transform = new NoVowels();
}else if ...

return transform.apply("Hello");
```

Polymorphism III

This may not seem to impressive, but what if the logic choosing the transformation is *not in our code*?

What if we have to pass a transform *into someone elses code*?

What if we need a *collection* of transformations?

Example VII

```
final SomeoneElses code = new SomeoneElsesCode();
// We have no idea what this could possibly return!
final Transform transform = code.getTransform();
// ...but we can use it anyway, because we know its interface!
```

```
return transform.apply("Hello");
```

Example VIII

In someone else's code:

```
public void setTransform(final Transform t){
  transform = t;
}
```

In our code:

```
//They have no idea what this is
final Transformer tx = new Reverse();
//...but they can still use it!
code.setTransform(tx);
```

Example IX

```
final List<Transform> txs = new ArrayList<>();
txs.add(new Reverse());
txs.add(new NoVowels());
//...
```

Side Quest

What are all of the angle brackets?

```
final List<Transform> txs = new ArrayList<>();
```

Side Quest II

I guess we're going to need to talk about generics.

We mentioned how an interface can be used to represent different classes by referring to them as instances of the interface instead.

Side Quest III

We can also do this with an *ancestor classes*, if all of the classes have a common ancestor. So, in the bad old days, a *List* would simply store *Object*s, because *Object* is a common ancestor for all other *classes*.

Raw Types

This led to problems:

```
final List txs = new ArrayList();
txs.add(new Reverse());
// Not an error, because it's an object
txs.add("Hello");
// Have to cast, because we need a Transform, not an Object
final Transform tx = (Transform)txs.get(1);
```

Generics

To fix this, **generics** were introduced to allow for more type safe code to be written:

```
final List<Transform> txs = new ArrayList<>();
txs.add(new Reverse());
//Now this is an error
txs.add("Hello");
//And a cast wouldn't be needed if the above didn't crash!
final Transform tx = txs.get(0);
```

Example X

Let's improve our *Transform* interface now:

```
public interface Transform<T,U> {
  U apply(final T input);
}
```

Now we're not limited to *String* transformations!

Example XI

```
public class Reverse implements Transform<String,String> {
  public String apply(final String input){
    //...
  }
}
```

Example XII

```
final List<Transform<String,String>> stringTransformers;
```

```
final List<Transform<Integer,Integer>> intTransformers;
final List<Transform<String,Integer>> stringToIntTransformers;
```

Can you feel the power??? Programming is so awesome.

Inheritance

Ok, let's dial things back a bit and get back to some basics.

What is inheritance? How does it work?

Inheritance II

Previously, we *implemented* an *interface*.

Sometimes, though, we want to *extend* a *class*.

Inheritance III

When a class B extends a class A, then the class B will **inherit** the methods and fields of class A.

Example XIII

```
public class ReReverse extends Reverse {

}
```

Example XIV

What will the following do? Will it compile?

```
final Reverse r1 = new ReReverse();
System.out.println(r1.apply("hello");
```

Example XIV

```
public class ReReverse extends Reverse {
  public String apply(final String input){
    final String reversed = super.apply(input);
    final String reReversed = super.apply(reveresed);
    return reversed;
```

```
  }
}
```

Super?

Example XV

What will the following do?

```
final Reverse r1 = new ReReverse();
System.out.println(r1.apply("hello");
final ReReverse r2 = new Reverse();
System.out.println(r2.apply("hello");
final Transform t = new ReReverse();
System.out.println("hello");
```

Polymorphism IV

When we declare a variable's type, that tells us what fields and methods *must* exist, but it might not be the *precise* type of the value bound to that variable.

(The actual value *may* have more fields and methods!)

Polymorphism V

When we call a method declared in the *parent* class, the JVM will determine, at run time, what the *actual* type of the value is, and will use this information to select the correct implementation of the called method.

What is the correct implementation?

Polymorphism VI

If the child class has *not* **overridden** the method, then the parent class will be checked for an implementation.

If the child class *has* **overriden** the method, then the implementation in the child class is chosen.

Remember, though, that you can have several levels of inheritance!

Access Modifiers

Don't forget that access modifiers affect what fields and methods are visible to children!

If, for example, a parent class declares a *private* field, the child can not access it!

Access Modifiers II

	none	private	protected	public
class	Y	Y	Y	Y
package	Y	N	Y	Y
subclass	N	N	Y	Y
world	N	N	N	Y

Spooky!

A common mistake students (and professionals!) make is to assume that the access modifiers apply to *instances* of a class. **This is not true!**

For instance, a **private** field in class A is visible ot **every instance** of A!

Example XVI

For example, this code is perfectly valid!

```
public class A {
  private final int x;

  //...

  public void compareXs(final A anotherA){
    // works even though x is private!
    return x==anotherA.x;
  }
}
```

Abstract Classes

Let's circle back around to some of the basics:

An **interface** (sort of) only provides method signatures.

A **class** must be **completely** implemented.

An **abstract class** is when you want something in between.

Example XVII

```
public abstract class Censor
                      implements Transform<String,String> {
  public abstract List<String> getBadWords();

  public String apply(String input){
    for(final String word : badWords){
      input = input.replace(word, "!!!")
    }
    return input;
  }
}
```

Now we can create different variations by subclassing and implementing *getBadWords*.

Inheritance IV

What if we want to inherit from two classes?

Inheritance V

You can't!

You can *implement* several interfaces, but you can only *extend* one class (concrete or abstract).

However...

Default Methods

You can now, actually, provide "default" implementations of methods in interfaces.

Which means that you can often accomplish what you were trying to do with two parent classes by instead implementing two interfaces with default methods.

Default Methods II

There are, however, limitations. First, remember that you can only store **static, constant** fields in an interface. If you need instance data for your default methods, you are out of luck.

Default Methods III

You can, though, add a *getter* to the interface for the instance data, and use that in your default method.

Then implementers will declare their own instance fields and implements the getter.

Example XVII

```
public interface Censor extends Transform<String,String> {
  List<String> getBadWords();

  default String apply(String input){
    for(final String bad : getBadWords()){
      input = input.replace(bad, "!!!");
    }
    return input;
  }
}
```

Implement or Extend

Another common mistake is to try and *implement* an interface in another interface. This makes no sense, because you aren't (generally) providing an implementation!

So, interfaces **extend** other interfaces!

Multiple Interfaces

An interesting use for multiple interfaces is to create an interface for each kind of behavior that might be needed, and mixing them together.

Example XVIII

```
public interface Labeled {
  public String getLabel();
}

public interface Logged {
  public List<String> getLog();
}
```

Example XVIV

```
public class Reverse
       implements Transform<String,String>, Labeled, Logged {
  private final List<String> log = new ArrayList<>();

  public String getLabel(){return "Reverse (Transform)";}
  public List<String> getLog(){return log;}

  public String apply(final input){
    log.add("Starting transform of input: " + input);
    //...
    log.add("Transform complete:" + result);
    return result;
  }
}
```

Multiple Interfaces II

So, in addition to being able to use this class wherever we need a *Transform*, we can use it anywhere we expect, for example, instances with a label.

Multiple Interfaces III

For example, imagine a UI where we can select operations to perform. Some might be transformers, other might not be, but we can put them all in a common UI widget with a nicely displayed label if they all implement the *Labeled* interface!

instanceof

At runtime, though, we may not know what interfaces are implemented, so we will need to check using the *instanceof* operator.

Example XIX

```
public void doTransform(final Transform<String,String> t){
  final String input = getInput();
  final String output = t.apply(input);
  if(t instanceof Logged){
    final Logged logged = (Logged) t;
    printLog(logged.getLog());
  }
  writeOutput(output);
}
```

Appendix: Lambda Expressions

Lambda Expressions

In this appendix, we will discuss interfaces, anonymous classes, and lambda expressions which are becoming a more commonly used feature in Java (and other languages) and allow for writing incredibly succinct, clear code.

Interfaces, Again

In order to understand lambda expressions, we will start with interfaces and slowly work our way to proper lambda expressions.

An Interface

Let's use the *Transform* interface from our Object Oriented review:

```
public interface Transform {
  String apply(final input String);
}
```

Implementations

We can use this interface to *implement* types of *Transformer*s.

```
public class ToCaps implements Transform {
  public String apply(final String input){
    return input.toUpperCase();
  }
}
```

Instances

Once we have a concrete class defined, we would typically create and use instances as follows:

```
final Transform tx = new ToCaps();
final String result = tx.apply("hello");
return result;
```

Anonymous Classes

Sometimes, though, we will only be creating instances for a class in *one location* in our code. Creating another file and another class is overkill, so instead we could use an anonymous class.

Anonymous Classes II

```
final Transform tx = new Transform(){
  public String apply(final String input){
    return input.toUpperCase();
  }
};
final String result = tx.apply("hello");
return result;
```

Anonymous Classes III

Basically, this allows us to declare the same class **inline**.

It's "anonymous," because it no longer has a name.

(it doesn't need one, because we are only referencing this class here!)

Going Further

If you look at the declaration of the anonymous class, you see that we have declared the variable to be a *Transform*, so in theory, we shouldn't have to tell the compiler we are creating a *Transform*.

Going Further II

This won't actually compile, but you should be able to see how the compiler *could* be implemented so that it would.

```
final Transform tx = {
  public String apply(final String input){
    return input.toUpperCase();
  }
};
final String result = tx.apply("hello");
return result;
```

Going Further III

Also, since there is only one method in the *Transform* interface, it seems like we should also be able to leave the method name, parameter type, return type, and access modifier out.

```
final Transform tx = {
  (input){
    return input.toUpperCase();
  }
};

final String result = tx.apply("hello");
return result;
```

Lambda Expression

Now, if we just eliminate unnecessary braces and tweak the syntax a bit, we can arrive at the following, which **will** compile.

```
final Transform tx = (input)->{
    return input.toUpperCase();
  };

final String result = tx.apply("hello");
return result;
```

Lambda Expression II

In this particular case, since there is only a single statement in the method body, we can be even more concise.

```
final Transform tx = (input)->input.toUpperCase();

final String result = tx.apply("hello");
return result;
```

Lambda Expressions III

And since there is only a single parameter, we can do a little better.

```
final Transform tx = input->input.toUpperCase();

final String result = tx.apply("hello");
return result;
```

Lambda Expressions IV

This may not seem that interesting, but let's look at a more complex example:

```
public class Transformer {
  private final Transform transform;
  public Transformer(final Transform tx){
    this.transform = tx;
  }

  public List<String> transform(final Source source){
    final List<String> output = new ArrayList<>();
    for(final String input : source){
      output.add(transform.apply(input));
    }
    return output;
  }
}
```

Lambda Expressions V

Now we can write really succinct code like:

```
private final Source = getSource();
final Transformer toUpper =
  new Transformer(s->s.toUpperCase());
```

```
toUpper.transform(source);
final Transformer toLower =
  new Transformer(s->s.toLowerCase());
toLower.transform(source);
```

GUI

...or maybe you have experienced the pain of writing a bunch of *ActionListener*s?

```
button.addActionListener(
  event->JOptionPane.showMessageDialog(this, "I was clicked!")
);
```

Even Better

In addition to *lambda expressions* we can also use **method references** to avoid even more boiler plate code!

Even Better II

Which gives us:

```
final Transformer toUpper =
      new Transformer(String::toUpperCase);
```

Instead of

```
final Transformer toUpper =
      new Transformer(s -> s.toUpperCase());
```

Functions

The most generalized version of this code would not even specify that a *Transform* is needed. Instead, the constructor would be:

```
public Transformer(Function<String,String> f){
  //...
}
```

Java provides *Function* and *BiFunction* interfaces to represent *any* one argument or two argument method, class, etc.

Also

For zero argument functions, Java provides the *Supplier* interface:

```
final Supplier<String> s = ()->"Hello!";
```

And for a "function" with no return, the *Consumer* interface:

```
final Consumer<String> c = s->System.out.println(s);
```

And Finally

If you want to pass a computation that has no parameters and no output, then you can use the *Runnable* interface:

```
new Thread(
  ()->{
    while(true){
      System.out.println("JavaScript Sucks");
    }
  }
).start();
```

Closures

One other interesting thing you can do with *anonymous classes* and *lambda expressions* is use them to *capture* variables in their **closures**.

Closures II

```
public Transformer create(String message, int count){
  return new Transformer(input->{
    final StringBuilder sb = new StringBuilder();
    for(int i=0; i&ltcount; i++){
      sb.append(text);
    }
  });
}

final Transformer t = create("hi",4);
t.transform(source); //message and count are out of scope!?
```

Closures III

This works, because any variable referenced in the lambda expression is *captured* and will be accessible for the life of the expression!

Appendix: Don't Null

Billion-Dollar Mistake

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. [...] I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement.

Billion-Dollar Mistake II

In reality, 1 billion dollars is a huge **under**statement of the financial damage caused by null references.

Most developers spend a comically large amount of time chasing null pointers.

Billion-Dollar Mistake III

...but **you** don't have to!

This one secret will save you time and money–more than the cost of this course!

What's The Problem?

Many will tell you that null pointers aren't a problem if you're a good programmer.

```
final User user = getUser(10);
final String name;

if(user !=null){
  name = user.getName();
}else{
  name = "Not Found";
}
```

The Problems

Problem: Every method could, possibly, return a null value, so to be safe you need to check the result of **every** method.

The "good programmers" don't do this, though. They "know" which methods need to be tested…which is why they, too, get null pointer exceptions.

The Problems II

Problem: If you don't want to test every method, you need to read the documentation for methods to determine which ones can return null.

The "good programmers" don't do this either…which is why they, too, get null pointer exceptions.

The Problems III

Problem: If you *do* read the documentation, it is imperative that the documentation is correct and complete.

The "good programmers" don't write and update documentation sufficiently, though…which is why they, too, get null pointer exceptions.

The Good Programmers

Of course, the "good programmers" always have excuses for why their null pointer problems are not *their* fault and continue to have null pointer problems.

The Dumb Programmer

Meanwhile, the "incompetent" programmer acknowledges his or her inability to cope with complex systems and asks the compiler for help.

```
public Optional<User> getUser(int id){
  if(userExists(id)){
    return Optional.of(new User(id));
  }else{
    return Optional.empty();
  }
}
```

The Dumb Programmer II

Now, this won't even compile!

```
final User user = getUser();
```

The compiler complains, because the return value is not a *User*. But we need an *User*...

The Dumb Programmer III

```
final Optional<User> maybeUser = getUser(10);
final String name;
if(maybeUser.isPresent()){
  name = maybeUser.get().getName();
}else{
  name = "Not Found";
}
```

Did we just do the same thing as the good programmer?

The Difference

Sort of. There is one immediate difference, though: Now we are **forced by the compiler** to address the possibility of no value being returned.

This is already a big gain with only slightly more code needed than the good programmer's approach.

The Difference II

If your team is on board with **never** returning null values (which is significantly easier than avoiding null values returned from methods unexpectedly), then you can use **Optional** values to eliminate null pointer exceptions from your code.

Java Is Not Great

Java, unfortunately, does not prevent you from returning a null value even if the return type is an **Optional** value.

You should have used Haskell.

Bonus

Another bonus to this approach is that the return type itself **documents the fact that the method may not return a value**. Even better, if this documentation changes, the compiler will force you to rethink your use of the method!

What About Get()

Of course, you could just **assume** that the method returns a value and write:

```
final String name = getUser(10).get().getName();
```

Which isn't any better than

```
final String name = getUser(10).getName()
```

Using the old code.

Psychology

The big difference, though, is psychology. If you know the meaning of an **Optional** return type, and you decide that, even though this method is guaranteed to sometimes not return a value, you are going to roll the dice, then there is literally no programming language feature that can help you, because you willfully do bad things.

Psychology II

On the other hand, the developer who doesn't check for a null is often just being careless. There is no red flag being waived in his or her face about the certain danger.

So, **Optional** return types flip the responsibility around.

Psychology III

With null values, you have to be responsible enough to make sure you don't screw up.

With **Optional** values, you are almost forced to write good, safe code and have to male a conscious decision to do the bad thing.

(Some even argue that get() should not exist in the Optional class)

The Problems IV

Problem: Why are we returning a default value of "Not Found"? If the *User* does not exist, then why should the *name* exist? We could just return a `null`, but that's bad!

Instead, we should just return an **Optional** name!

A Mess

```
final Optional<User> maybeUser = getUser(10);
if(maybeUser.isPresent()){
  final User u = maybeUser.get();
  return Optional.of(u.getName());
}else{
  return Optional.empty();
}
```

Am I Trolling?

At this point you should be ready to revolt and assume I am trolling you.

Are you really, supposed to litter **Optional** wrappers throughout every line of code?

Improvement

No! Higher order functions will rescue us and make life beautiful.

The **Optional** class defines a method called **map** which accepts a **function** as its argument.

Improvement II

More specifically, for the type **Optional** of type **T**, it accepts a function that takes an object of type **T** and returns some other type of object.

Improvement III

Using this, we can write the following instead:

```
return getUser(10).map(u->u.getName());
```

If the **Optional** contains a value, the function is applied to it, resulting in an **Optional** containing an *String*. Otherwise, the whole thing is just an empty **Optional**.

The Dumb Programmer Wins

Now, we have obtained safety **and** simpler code. Good programmers and their null checks have been thoroughly defeated at this point.

Going Further

What if the *getName()* method itself returns an *Optional*

```
final Optional<String> result =
  getUser(10).map(u->u.getName());
```

This doesn't compile. Why?

Improvement IV

The **Optional** type also provides a method **flatMap** that takes a function which returns an **Optional** value and "flattens" the **Optional** layers so that there are no nested **Optional**s

Awesome

Now, we can write safe code in the face of missing values quite simply:

```
return getUser(10)
         .flatMap(u -> u.getName())
         .map(n -> n.toUpperCase())
         .filter(n -> n.equals("PHILLIP"))
         .map(n->"User: " + n)
```

Improvement V

We can do a little cleanup using method references

```
return getUser(10)
         .flatMap(User::getName)
         .map(String::toUpperCase)
         .filter(n -> n.equals("PHILLIP"))
         .map(n->"User: " + n)
```