Microsoft

Java API Design Best Practices

Jonathan Giles
Java Guy at Microsoft

jonathan.giles@microsoft.com
@JonathanGiles

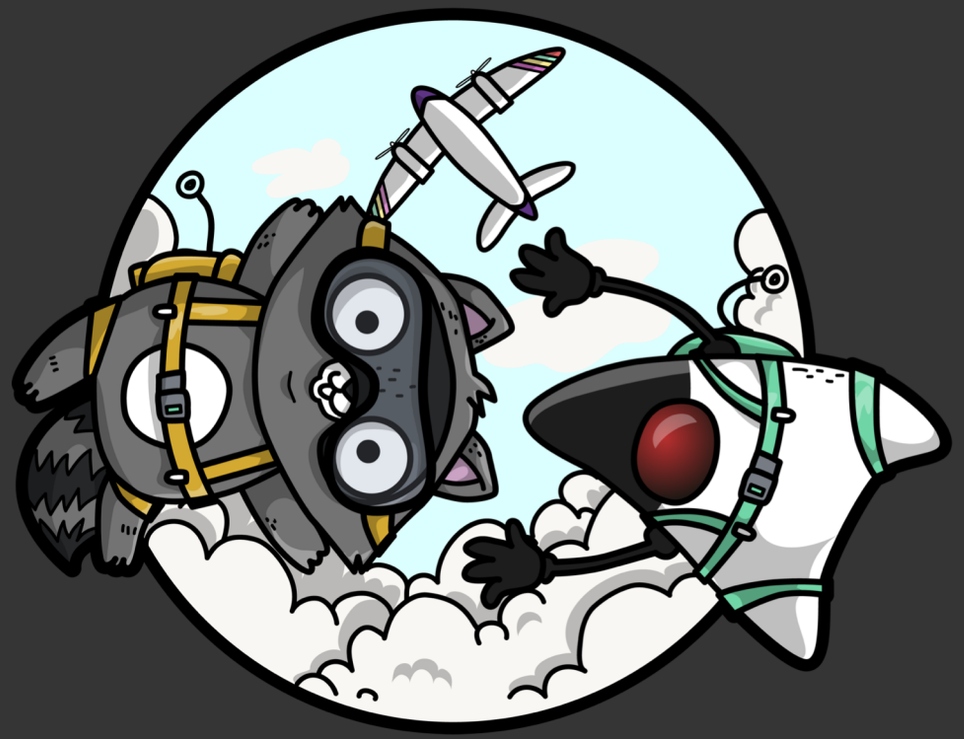
Hi There!

I'm Jonathan.
I used to work at Sun / Oracle on Java,
but now I work at Microsoft.

My passion is developer experience.
I care about API, documentation, and
anything that limits productivity.

Agenda

API Design Theory



Practical Advice

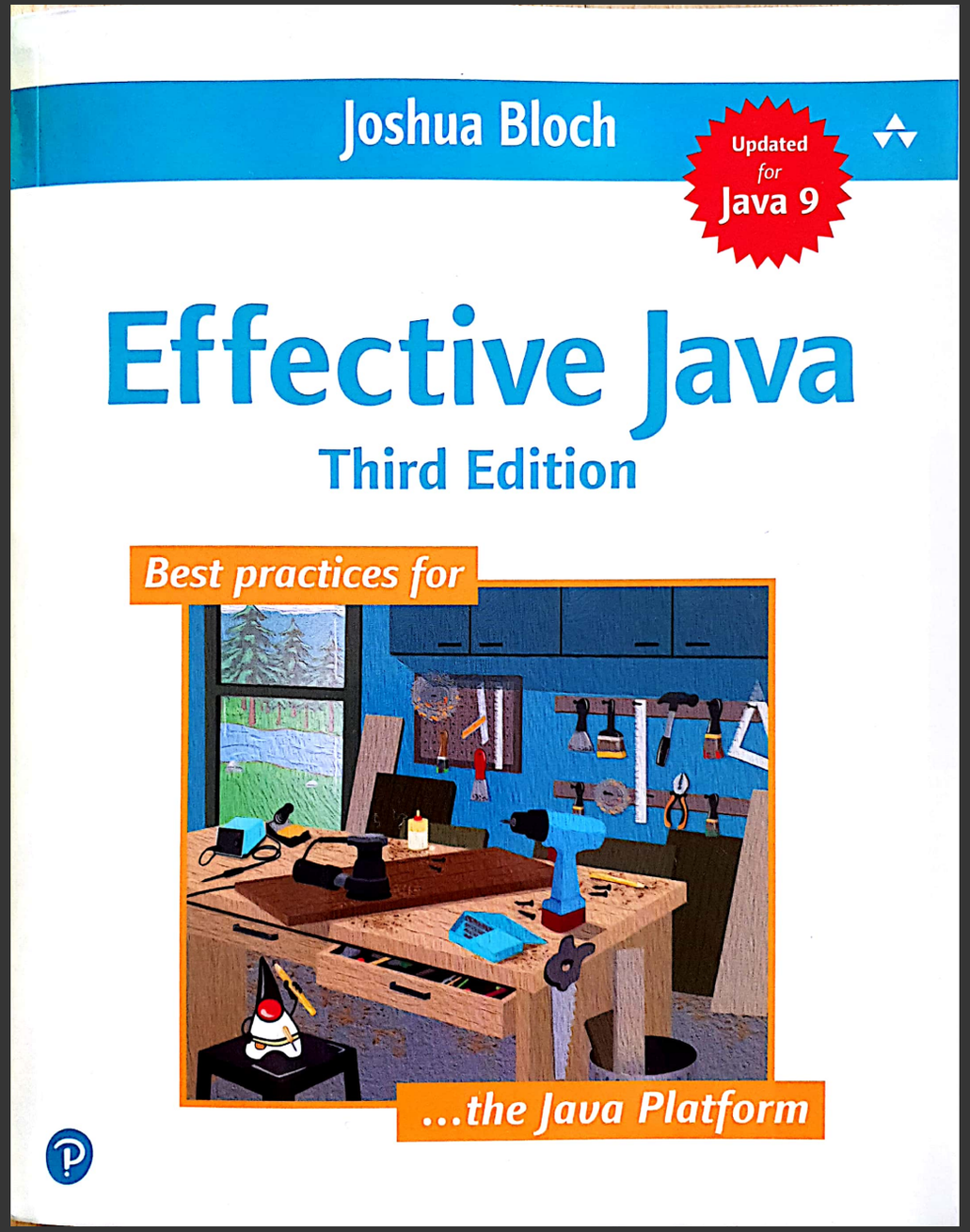
Effective Java 3rd Edition

Read this book!

A lot of the advice in this book is from my personal experiences, but it is also discussed in much more depth in this book.

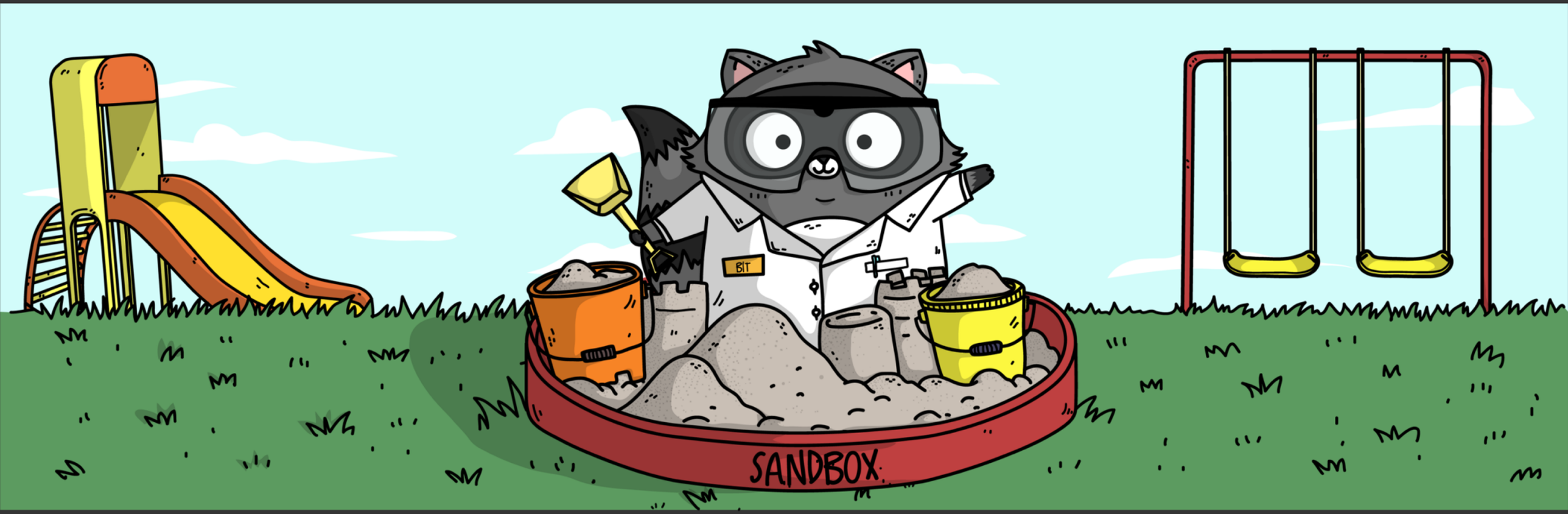
Further reading

Effective Java 3rd Edition is broken up into 90 items. Whenever I discuss a concept that is covered in the book, I will note the item number from the book.



API Design Theory

What Is API Design?



What Is API Design?

- An API is what a developer uses to achieve some task
  - It abstracts implementation, allowing us to work at a higher level of abstraction

- Key questions:
  - Who is the user of the API?
  - What are the goals of the user?

We are all API Designers

API Characteristics

- API has to be
  - Understandable
  - Well documented
  - Consistent
  - Fit for purpose
  - Restrained
  - Evolvable

API Characteristics - Understandable

- How do developers discover and make use of a new API?
  - An API should not be considered successful if a developer cannot intuitively understand how to use it.
    - External documentation (non-JavaDoc) should not be required ideally.
  - Object-orientation has made API discovery more difficult.

- Developers should consider the ‘entry points’ into their API.

API Characteristics - Consistency

- A good API should not surprise its users
- Consistency enables developers to intuit new API

API Characteristics - Consistency

- A minimal set of return types should be used
  - What to return for a collection? e.g. List / Collection / Iterator / Iterable / Stream
- If some methods are documented to not return null for a certain type, never return null for that type in any method

API Characteristics - Consistency

- Returning null enables NPE to crop up
  - Consistently use conventions to return non-null values instead

Return Type	Non-null Return Value
String	"" (empty string)
List / Set / Map / Iterator	Use Collections class, e.g. Collections.emptyList() / Collections.emptySet() / etc
Stream	Stream.empty()
Array	Return an empty, zero-length array
All other types	Consider using Optional (discussed later in presentation)

API Characteristics - Consistency

- Method naming patterns should be planned up-front
  - Establish a vocabulary to use repeatedly throughout the API for types, methods, arguments, constants, etc.
  - Method names like Type.of(), Type.valueOf(), Type.toXYZ(), Type.from(), etc. should be used consistently, and never mixed.

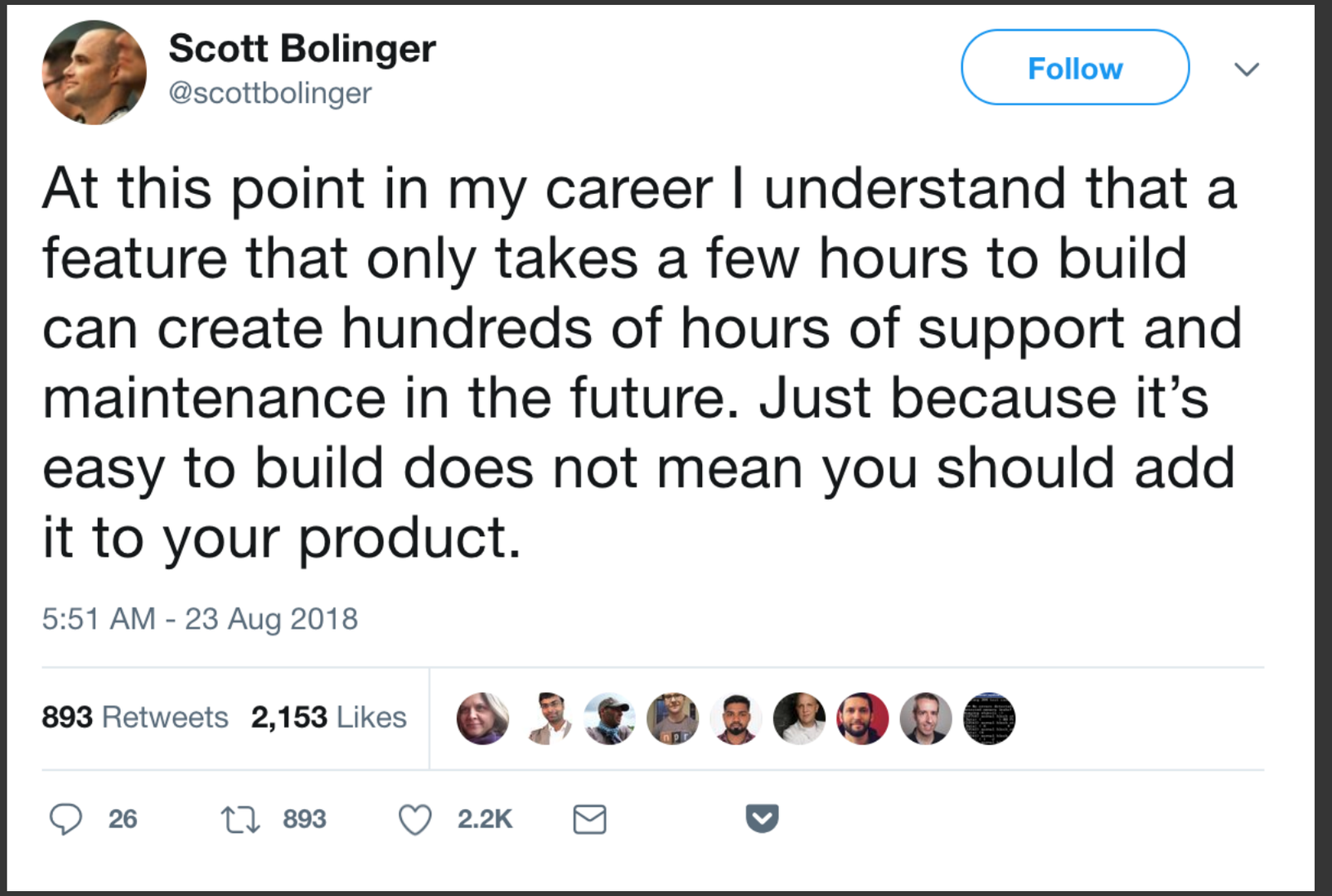
API Characteristics - Consistency

- Argument order should be consistent
  - If a method is overloaded, keep the order consistent whenever possible
- If the argument size becomes unwieldy, consider introducing argument objects
  - A class containing the values that would ordinarily go into the method argument.
  - This allows for better growth of the API over time, as the argument type can have more fields added with ease.

API Characteristics – Fit For Purpose

- In developing an API we must ensure that we target it at the right level for the intended user. This can be thought of in two ways:
  1. Do only one thing, and do it right.
  2. Understand your user and their goals.

API Characteristics - Restrained



API Characteristics - Restrained

- It is easy to think that we should make developer lives easier by having as much API as possible
- Two concerns:
  - Developer overload – too much API to easily understand how to use it
  - The more API we expose, the greater our maintenance burden

API Characteristics - Restrained

- Every API needs justification
- New API designers tend to favor maximal API designs
  - *"If I add this function, it'll save the user X lines of code"*
- My advice: invert this desire!
  - Force yourself to justify every public method
  - Ask yourself: "Does adding this increase the burden on me, as the API designer?"
  - This does not mean there should be no convenience API!

API Characteristics - Restrained

- Convenience API is important to a good API
  - e.g. List.of(..) or List.add(Object)
  - These convenience methods enable developers to save substantial amounts of code
- There is an important gut feeling to develop here:
  - What is the right amount of convenience?

API Characteristics - Restrained

- Our default position should be to make classes and public methods final
- Start with private modifiers, and increase visibility only after consideration
  - Fields should rarely be public
- Introduce protected API carefully
  - Before committing to it, write subclasses that use it

API Characteristics - Restrained

- Understand, and properly manage, implementation classes
- Two primary approaches
  1. Put implementation into packages under an 'impl' package
  2. Make impl classes 'package-private' (i.e. have no modifier on the class)
- When reviewing JavaDoc, make sure no implementation leaks out from public API!

API Characteristics - Restrained

- Your API is a contract
  - If you expose external dependencies, they become part of your contract
  - Be careful to only expose the bare minimum
- Consider whether the API should be exposed, or if you should expose a wrapper API instead

API Characteristics - Evolvable

- Our API contract should state our policy on backwards compatibility and deprecation
- Semantic Versioning: https://semver.org/
- Example:
  - Adding new API is acceptable, but removing or modifying existing API can only happen in a major release, after one release being deprecated

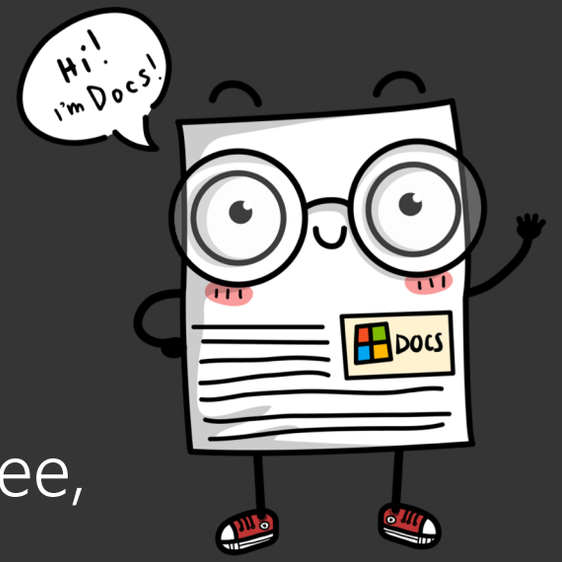
API Characteristics - Evolvable

- The ‘journey to 1.0.0’
  - API design is cheap
    - Spend cycles on it before committing to implementation
    - ‘Eat your own dog food’
- Projects have different breaking changes policies
  - Don’t feel overly locked-down – it depends on how important backwards compatibility is for your community

Eating Your Own Dog Food

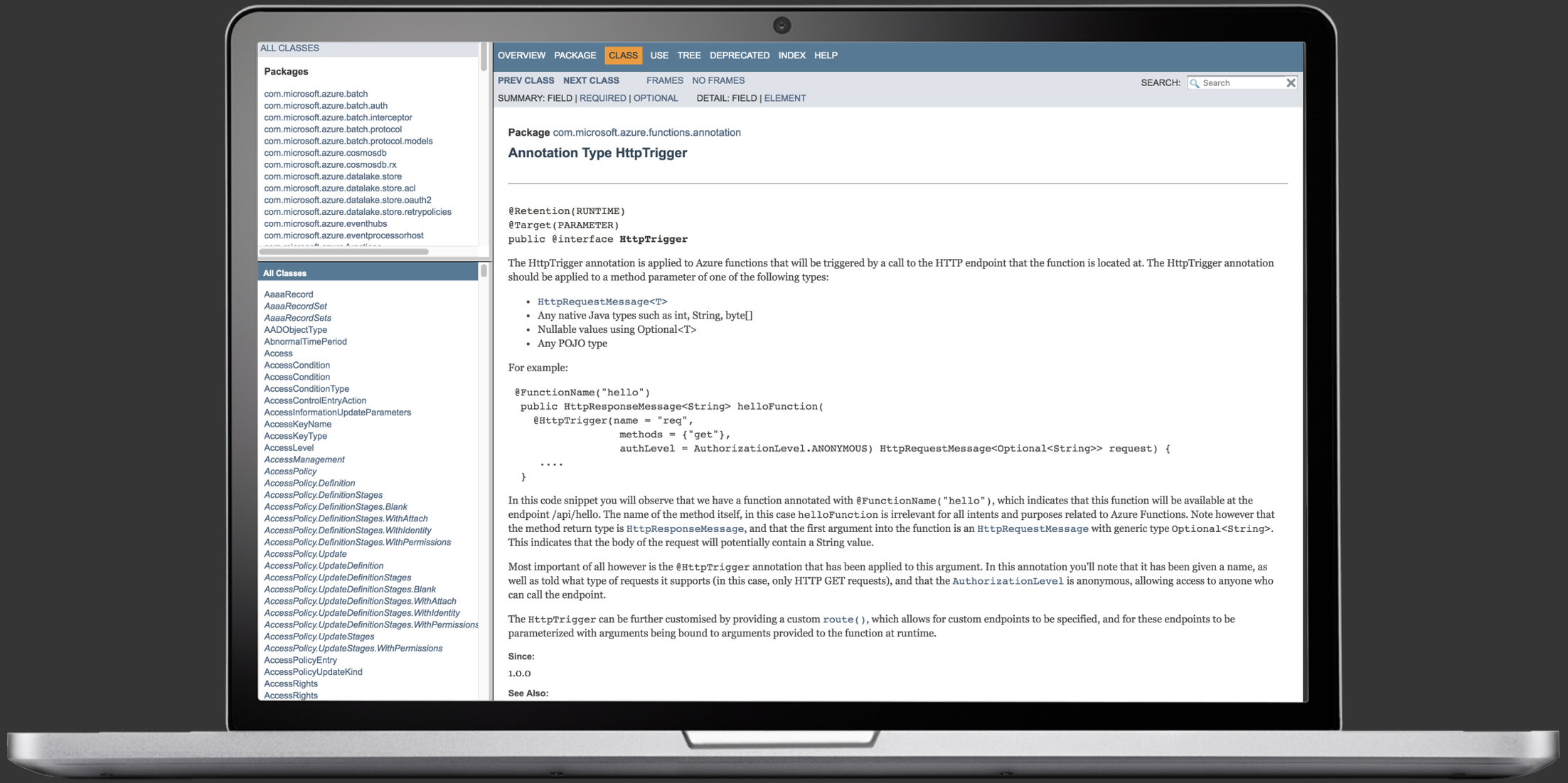
- Have developer empathy
  - See the problem domain from your users eyes
- Write sample code with your API and discuss it with real users
- Review sample code for
  - Unclear intentions
  - Duplicate, or redundant code
  - Abstraction is too low-level or too high-level

API Characteristics - Documentation

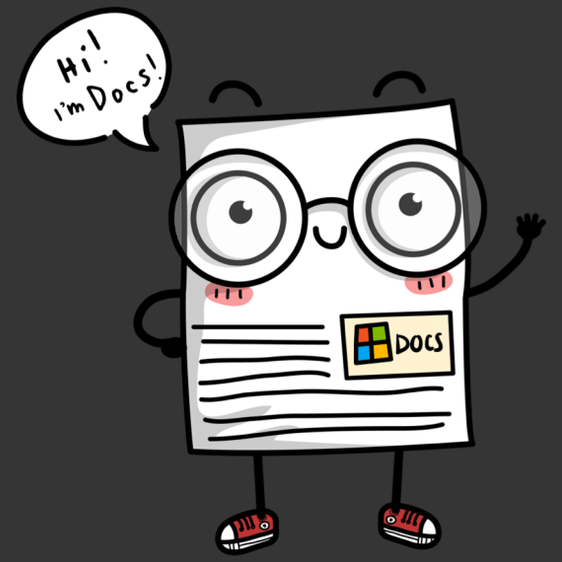


- Write quality JavaDoc
  - Make use of common 'annotations' to help readers (@see, @since, @link, etc).
  - Include small code snippets demonstrating how to use the class
    - These can be added as a result of user bug reports – to clarify how an API is used.

API Characteristics - Documentation



API Characteristics - Documentation



- Use JavaDoc to specify behavioral contracts
- For example:
  - Arrays.sort() method guarantees it is stable (equal elements are not reordered).
- It isn't right to specify this guarantee in the API
  - The JavaDoc details therefore form part of the API contract
- Behavioral contracts should be treated as API
  - Adding, changing, or removing them should be carefully considered

API Characteristics - Documentation

- JavaDoc is a great way to review API
  - Get in the habit of generating the HTML output and reviewing
    - Look for things that don't feel right
    - Look for missing or incorrect JavaDoc
    - Look for unintentional API

API Characteristics - Documentation

- Do not include ‘negative’ examples in your code
  - e.g. “Here is some code you should never write: …”
  - Users don’t read the text before or after code snippets
  - A large proportion of bug reports in your next release will be about this code node working right.

API Characteristics - Documentation

A warning about inserting Nodes into the ComboBox items list

ComboBox allows for the items list to contain elements of any type, including `Node` instances. Putting nodes into the items list is **strongly not recommended**. This is because the default `cell factory` simply inserts Node items directly into the cell, including in the ComboBox 'button' area too. Because the scenegraph only allows for Nodes to be in one place at a time, this means that when an item is selected it becomes removed from the ComboBox list, and becomes visible in the button area. When selection changes the previously selected item returns to the list and the new selection is removed.

The recommended approach, rather than inserting Node instances into the items list, is to put the relevant information into the ComboBox, and then provide a custom `cell factory`. For example, rather than use the following code:

```
ComboBox<Rectangle> cmb = new ComboBox<Rectangle>();
cmb.getItems().addAll(
    new Rectangle(10, 10, Color.RED),
    new Rectangle(10, 10, Color.GREEN),
    new Rectangle(10, 10, Color.BLUE));
```

You should do the following:

```
ComboBox<Color> cmb = new ComboBox<Color>();
cmb.getItems().addAll(
    Color.RED,
    Color.GREEN,
    Color.BLUE);

cmb.setCellFactory(new Callback<ListView<Color>, ListCell<Color>>() {
    @Override public ListCell<Color> call(ListView<Color> p) {
        return new ListCell<Color>() {
            private final Rectangle rectangle;
            {
                setContentDisplay(ContentDisplay.GRAPHIC_ONLY);
                rectangle = new Rectangle(10, 10);
            }

            @Override protected void updateItem(Color item, boolean empty) {
                super.updateItem(item, empty);

                if (item == null || empty) {
                    setGraphic(null);
                } else {
                    rectangle.setFill(item);
                    setGraphic(rectangle);
                }
            }
        };
    }
});
```


Admittedly the above approach is far more verbose, but it offers the required functionality without encountering the scenegraph constraints.

Team Consensus

- Create a team-wide cheat sheet
  - Share with new hires
  - Ensures consistency
  - Have a way to enable team members to give feedback

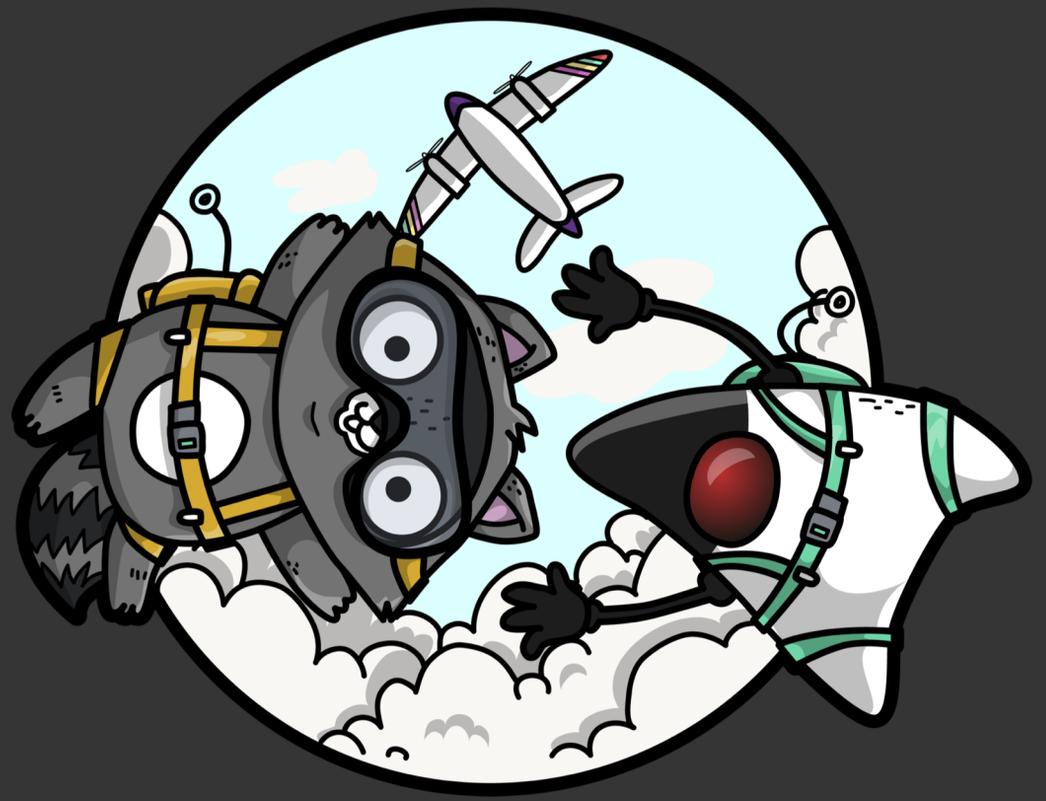
Our goal: getting everyone moving in the same direction

In conclusion:

There is no magical process to API design.

API design is an art,
and like art,
becomes easier with practice

Practical Advice



Tip 1: Static Factories

- Static factories offer three benefits over constructors:
  1. Ability to be named (i.e. constructors must be the class name)
  2. They do not require a new instance to be created
  3. Ability to return subclasses

Tip 1: Static Factories

We've been using them all along in the JDK:

```
public static Boolean valueOf(boolean b) {
  return b ? Boolean.TRUE : Boolean.FALSE;
}
```

and there are always new static factories being added, e.g.:

```
static <E> List<E> of();
static <E> List<E> of(E e1);
static <E> List<E> of(E e1, E e2);
// ....and so on (there are 12 overloaded versions of this method!)

static <E> List<E> of(E... elems);
```

Tip 1: Static Factories

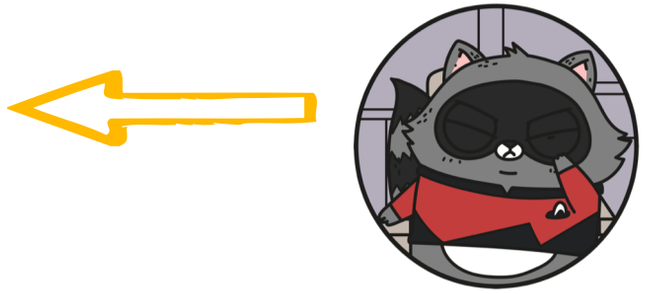
```
public class RandomIntGenerator {
  private final int min;
  private final int max;

  public int next() { … }

  public RandomIntGenerator(int min, int max) {
    this.min = min;
    this.max = max;
  }

  public RandomIntGenerator(int min) {
    this(min, Integer.MAX_VALUE);
  }

  public RandomIntGenerator(int max) {
    this(Integer.MIN_VALUE, max);
  }
}
```



Duplicate method

https://jlordiales.me/2012/12/26/static-factory-methods-vs-traditional-constructors/

Tip 1: Static Factories

```
public class RandomIntGenerator {
  private final int min;
  private final int max;

  private RandomIntGenerator(int min, int max) {
    this.min = min;
    this.max = max;
  }

  public static RandomIntGenerator between(int min, int max) {
    return new RandomIntGenerator(min, max);
  }

  public static RandomIntGenerator biggerThan(int min) {
    return new RandomIntGenerator(min, Integer.MAX_VALUE);
  }

  public static RandomIntGenerator smallerThan(int max) {
    return new RandomIntGenerator(Integer.MIN_VALUE, max);
  }

  public int next() {...}
}
```

https://jlordiales.me/2012/12/26/static-factory-methods-vs-traditional-constructors/

Tip 1: Static Factories

- Contentious…
  - https://dzone.com/articles/constructors-or-static-factory-methods
- As with all advice today – form your own opinions
  - Even if you disagree, follow the spirit of the advice:
    - Developer empathy
    - API quality
    - High design standards

Tip 2: Understand When To Use Optional

- Java 8 introduced Optional as a way of lessening NPE
  - An Optional<T> contains one element of type T, or is empty
- Optional is best used in select cases when:
  - A result might not be able to be returned
  - The API consumer has to perform some different action in this case
- Optional provides a number of convenience methods

Tip 2: Understand When To Use Optional

```
// getFastest returns Optional<Car>, but if the cars list is empty, it
// returns Optional.empty(). In this case, we can choose to map this to an
// invalid value.
Car fastestCar = getFastest(cars).orElse(Car.INVALID);

// If the orElse case is expensive to calculate, we can also use a Supplier
// to only generate the alternate value if the Optional is empty
Car fastestCar = getFastest(cars).orElseGet(() -> searchTheWeb());

// We could alternatively throw an exception
Car fastestCar = getFastest(cars).orElseThrow(MissingCarsException::new);

// We can also provide a lambda expression to operate on the value, if it
// is not empty
getFastest(cars).ifPresent(this::raceCar)
```

Tip 2: Understand When To Use Optional

```
// Whilst it is ok to call get() directly on an Optional, you risk a
// NoSuchElementException if it is empty. You can wrap it with an
// isPresent() call as shown below, but if your API is commonly used like
// this, it suggests that Optional might not be the right return type
Optional<Car> result = getFastest(cars);
if (result.isPresent()) {
  result.get().startCarRace();
}
```

Tip 2: Understand When To Use Optional

```
// Some people just want to see the world burn
public Optional<Car> getFastest(List<Car> cars) {
  if (cars == null || cars.isEmpty()) {
    return null;
  }
  ...
}
```

Tip 2: Understand When To Use Optional

- As discussed in the previous tip, don't use Optional in all cases
  - Do not do Optional<Collection<T>>, simply return an empty Collection<T> when there are no elements.

Tip 3: Become Familiar With java.util.function

- It's very enticing to write your own @FunctionalInterface's
- Before doing this – spend time becoming familiar with the interfaces in java.util.function
  - In here you'll find 43 standard functional interfaces
  - Can be broken down into six categories

Tip 3: Become Familiar With java.util.function

Interface	Signature	Summary
UnaryOperator<T>	T apply(T t)	`UnaryOperator<T> extends Function<T,T>`
BinaryOperator<T>	T apply(T t1, T t2)	`BinaryOperator<T> extends BiFunction<T,T,T>`
Predicate<T>	boolean test(T t)	`Takes a T, returns a primitive boolean value`
Function<T,R>	R apply(T t)	`Takes a T, returns an object of type R`
Supplier<T>	T get()	`Takes no argument, returns an object of type T`
Consumer<T>	void accept(T t)	`Takes a T, returns nothing`

Tip 3: Become Familiar With java.util.function

- In some cases, the existing interfaces do not meet our needs
  - Their name is not descriptive
  - You want to add default methods to the interface

- Use the @FunctionalInterface annotation
  - This informs devs and the compiler the interface is designed for lambdas
  - The interface will only compile if it has one abstract method

Tip 4: Support Lambdas

- When designing API, consider if it can support lambdas
- Requirement for lambdas:
  - The argument type must be a 'functional interface' (or abstract class)
    - A single abstract method

Tip 4: Support Lambdas

Java Swing UI Toolkit:

```
JButton btn = new JButton(“Click Me”);
btn.addMouseListener(new MouseListener() {
  @Override public void mouseReleased(MouseEvent e) { .. }
  @Override public void mousePressed(MouseEvent e) { .. }
  @Override public void mouseExited(MouseEvent e) { .. }
  @Override public void mouseEntered(MouseEvent e) { .. }
  @Override public void mouseClicked(MouseEvent e) { .. }
});
```

Tip 4: Support Lambdas

Java Swing UI Toolkit:

```
JButton btn = new JButton(“Click Me”);
btn.addMouseListener(new MouseAdapter() {
  @Override public void mouseClicked(MouseEvent e) { .. }
});
```

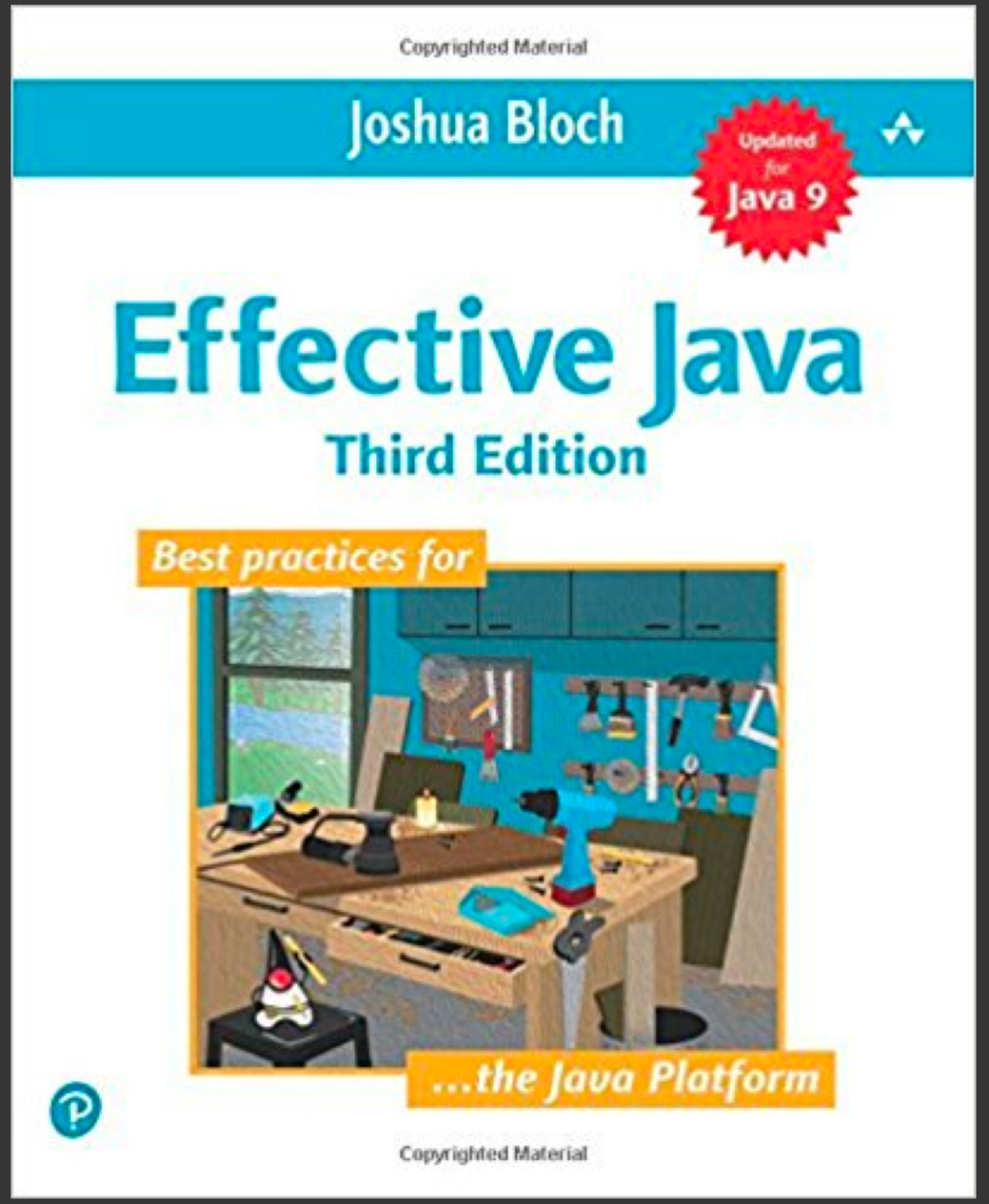
Tip 4: Support Lambdas

JavaFX UI Toolkit:

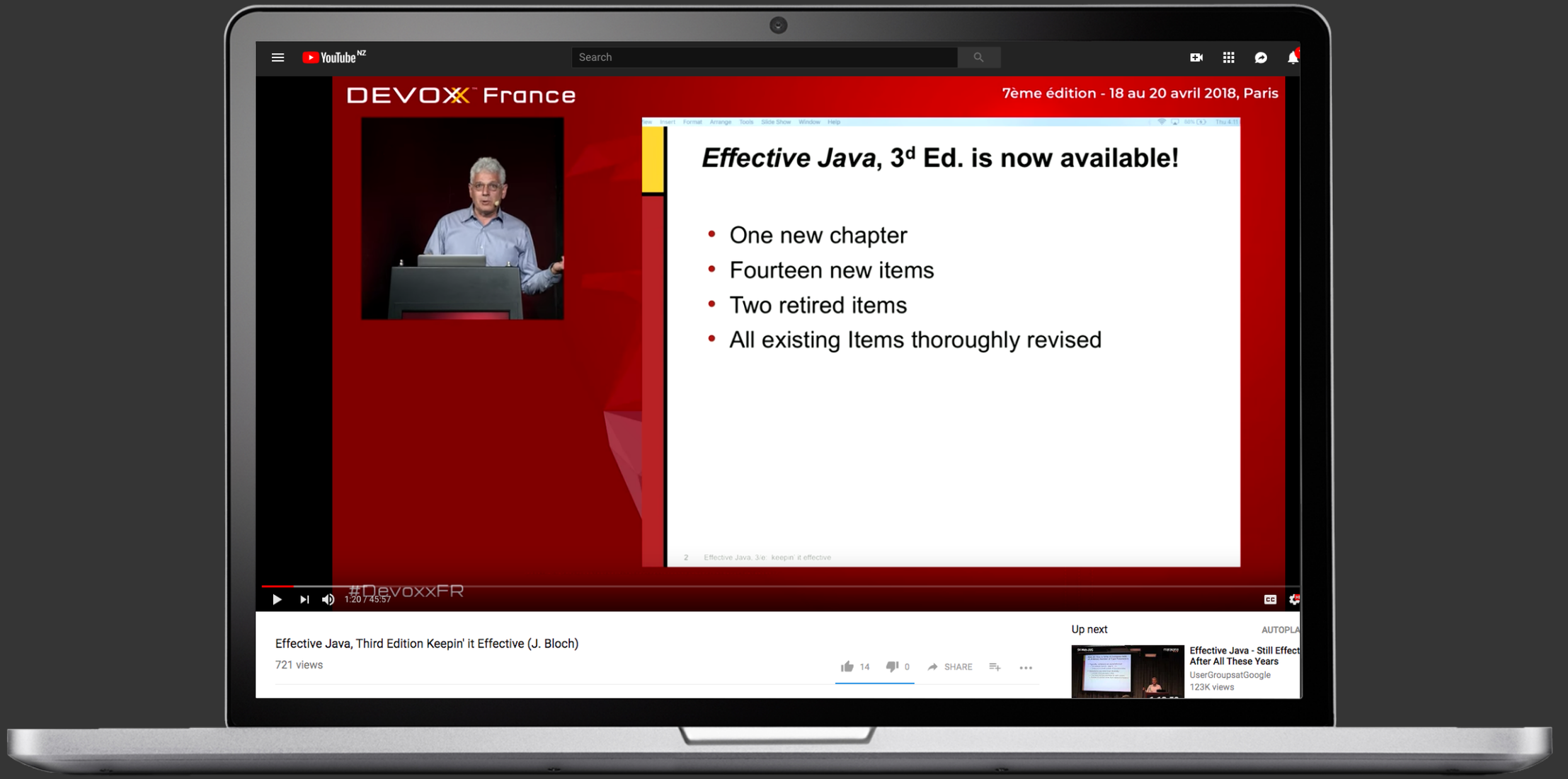
```
Rectangle rect = new Rectangle();
rect.setOnMouseClicked(new EventHandler<MouseEvent>() {
  @Override public void handle(MouseEvent e) {
    print(e);
  }
});
```

```
Rectangle rect = new Rectangle();
rect.setOnMouseClicked(e -> print(e));
```

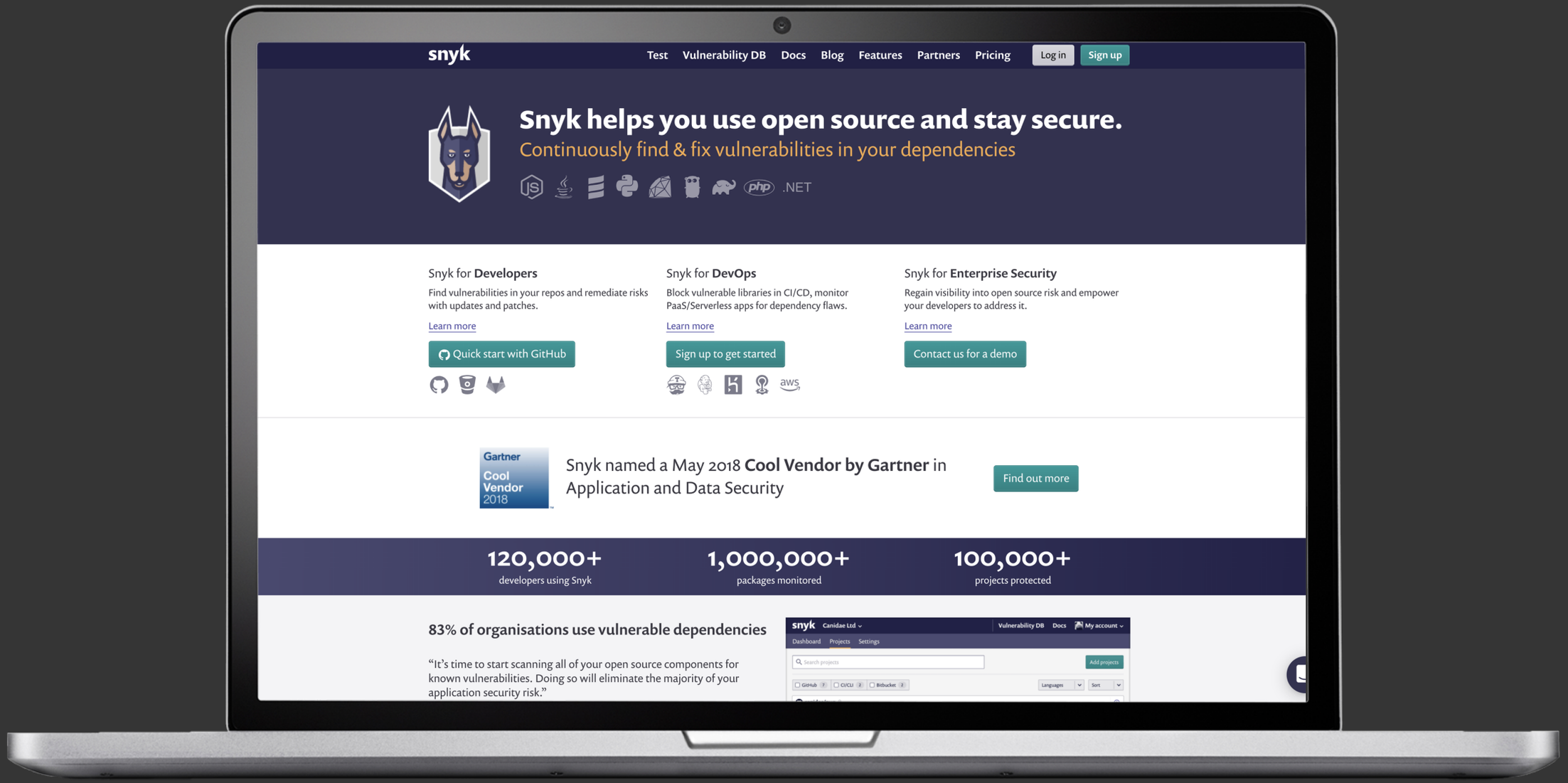
Resources


Copyrighted Material
Joshua Bloch
Updated for Java 9
Effective Java
Third Edition
Best practices for
...the Java Platform
Copyrighted Material

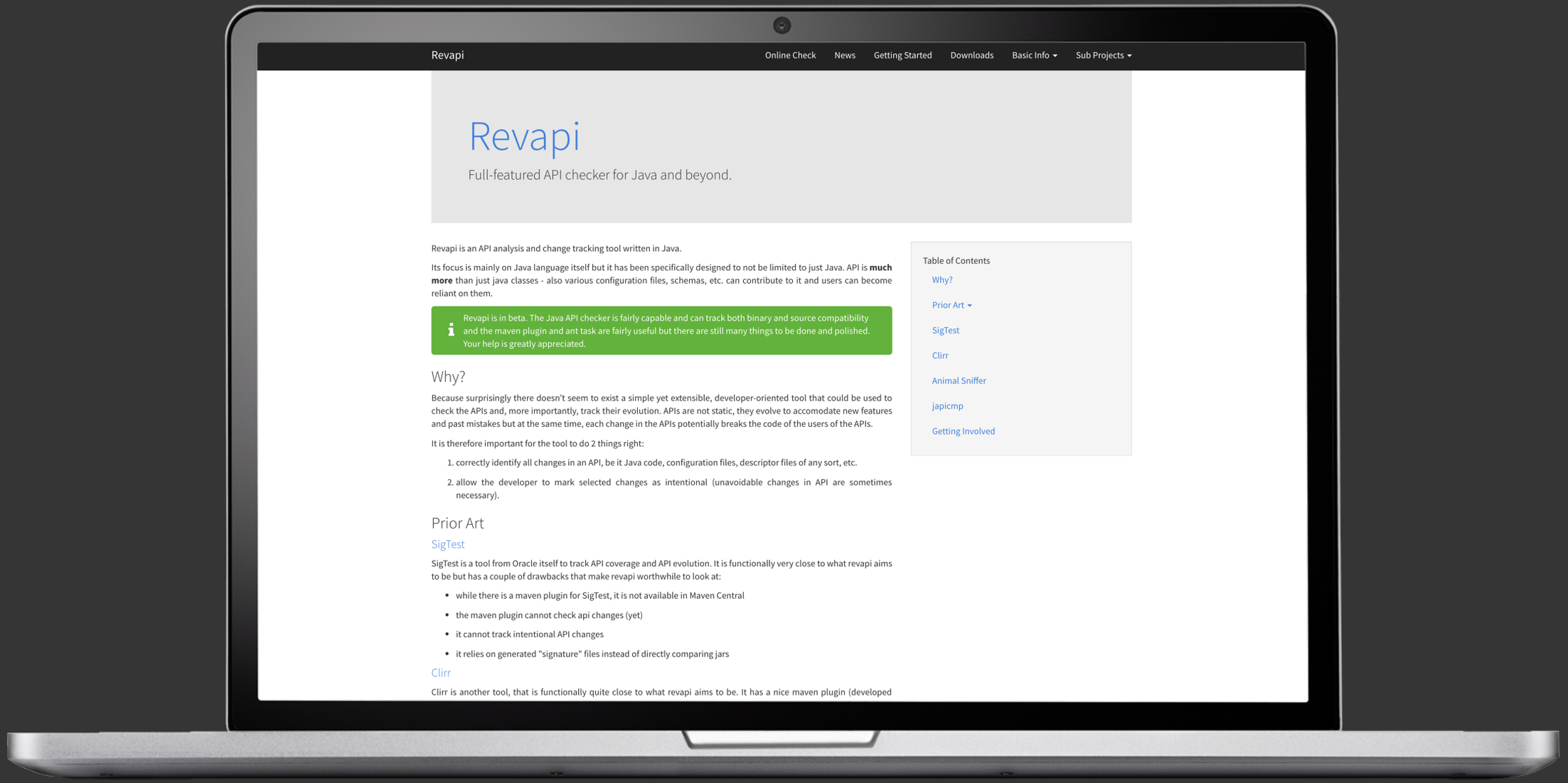
YouTube


YouTube NZ
Search
DEVOXX France
7ème édition - 18 au 20 avril 2018, Paris
Effective Java, 3d Ed. is now available!
One new chapter
Fourteen new items
Two retired items
All existing Items thoroughly revised
#DevoxxFR
1:20 / 45:57
Effective Java, Third Edition Keepin' it Effective (J. Bloch)
721 views
14
0
SHARE
Up next
Effective Java - Still Effect
After All These Years
UserGroupsatGoogle
123K views

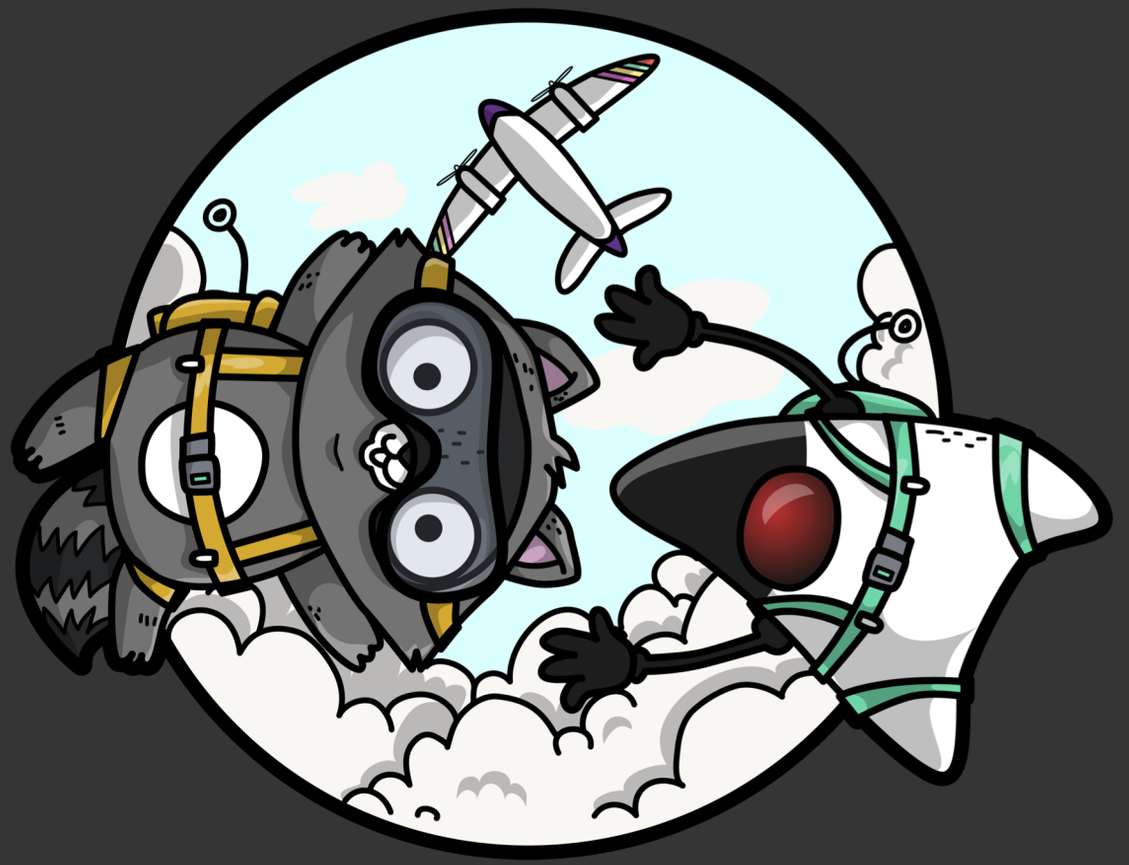
Snyk – http://snyk.io



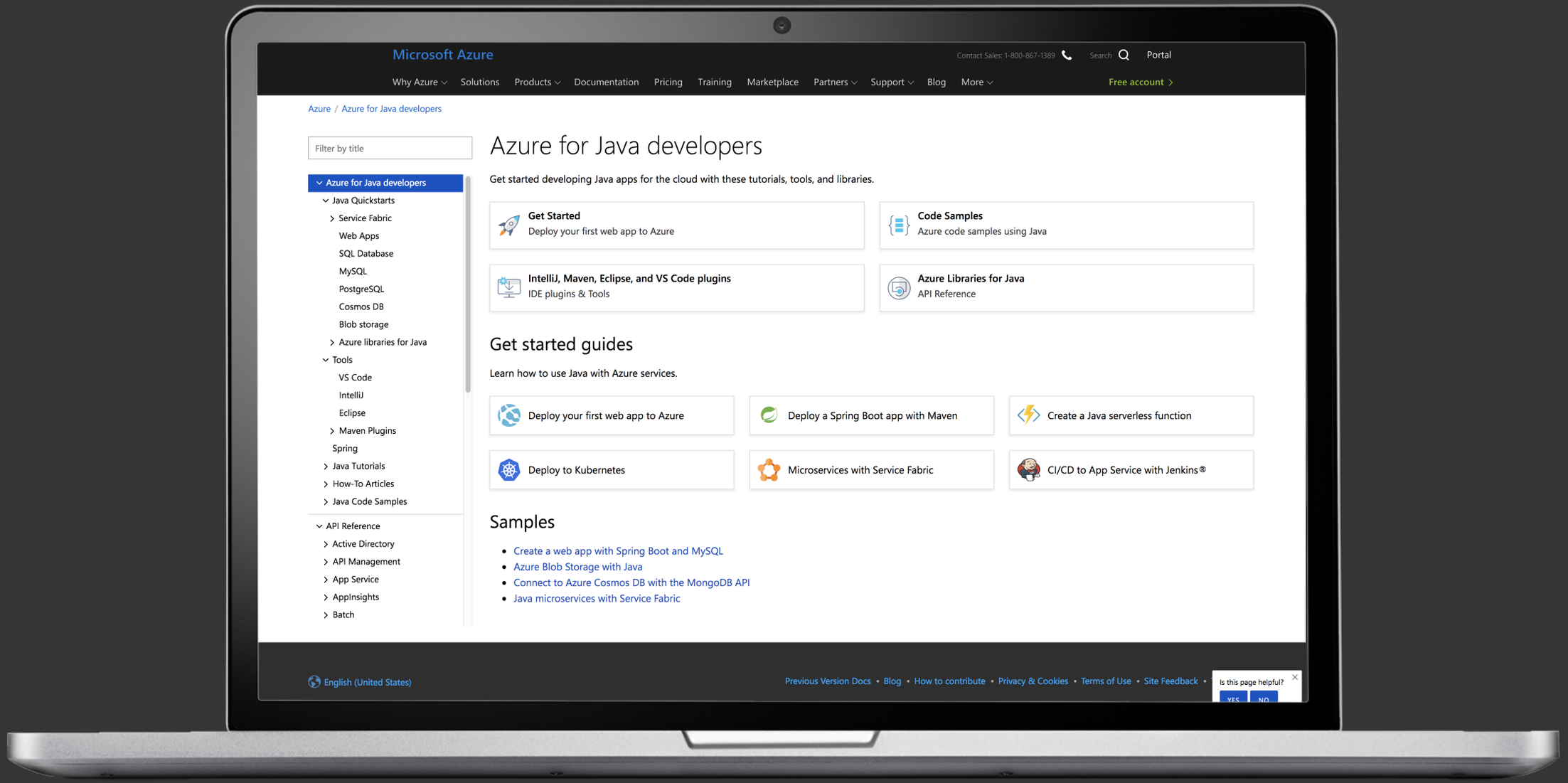
RevAPI – http://revapi.org



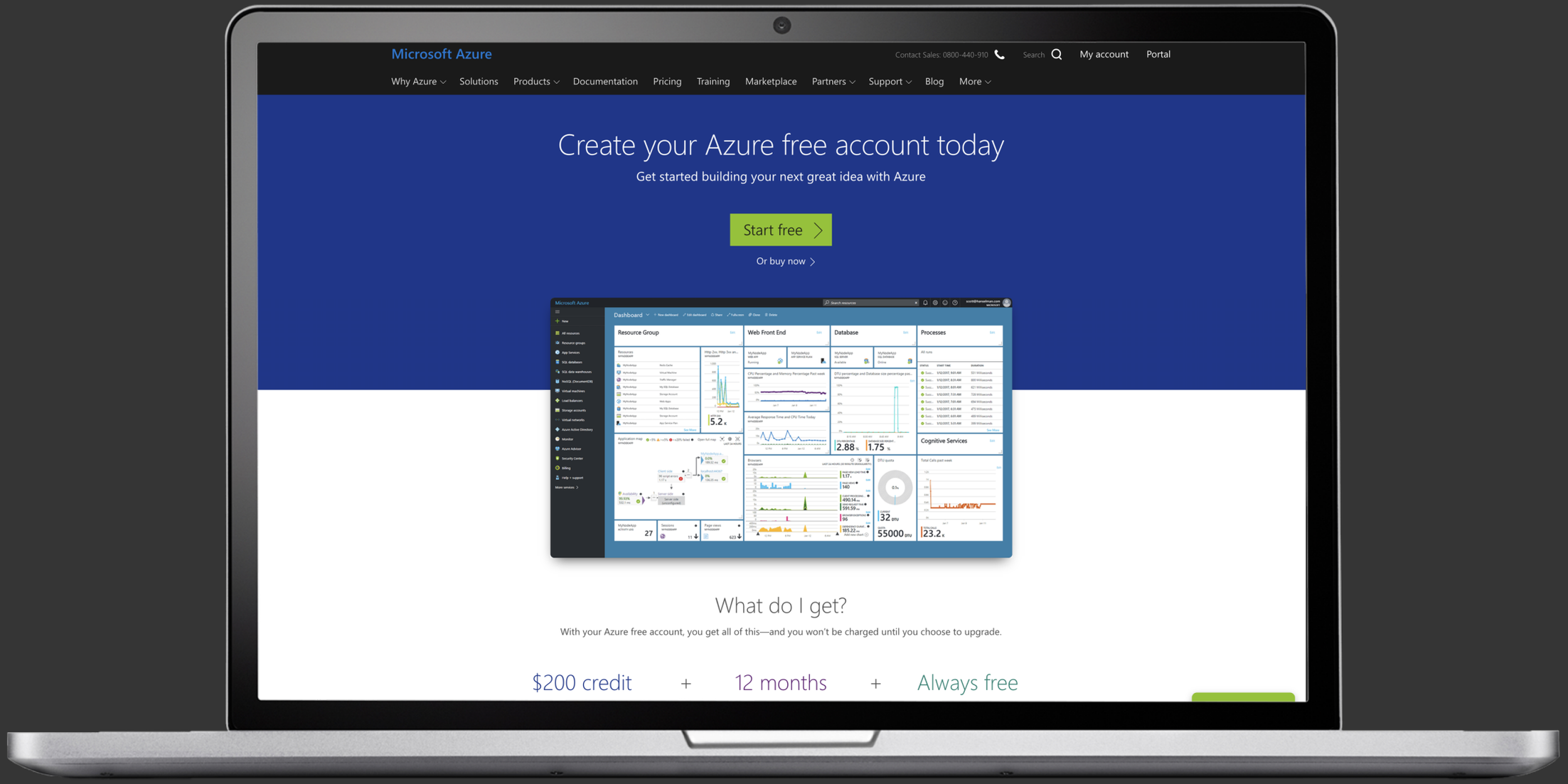
Useful Links



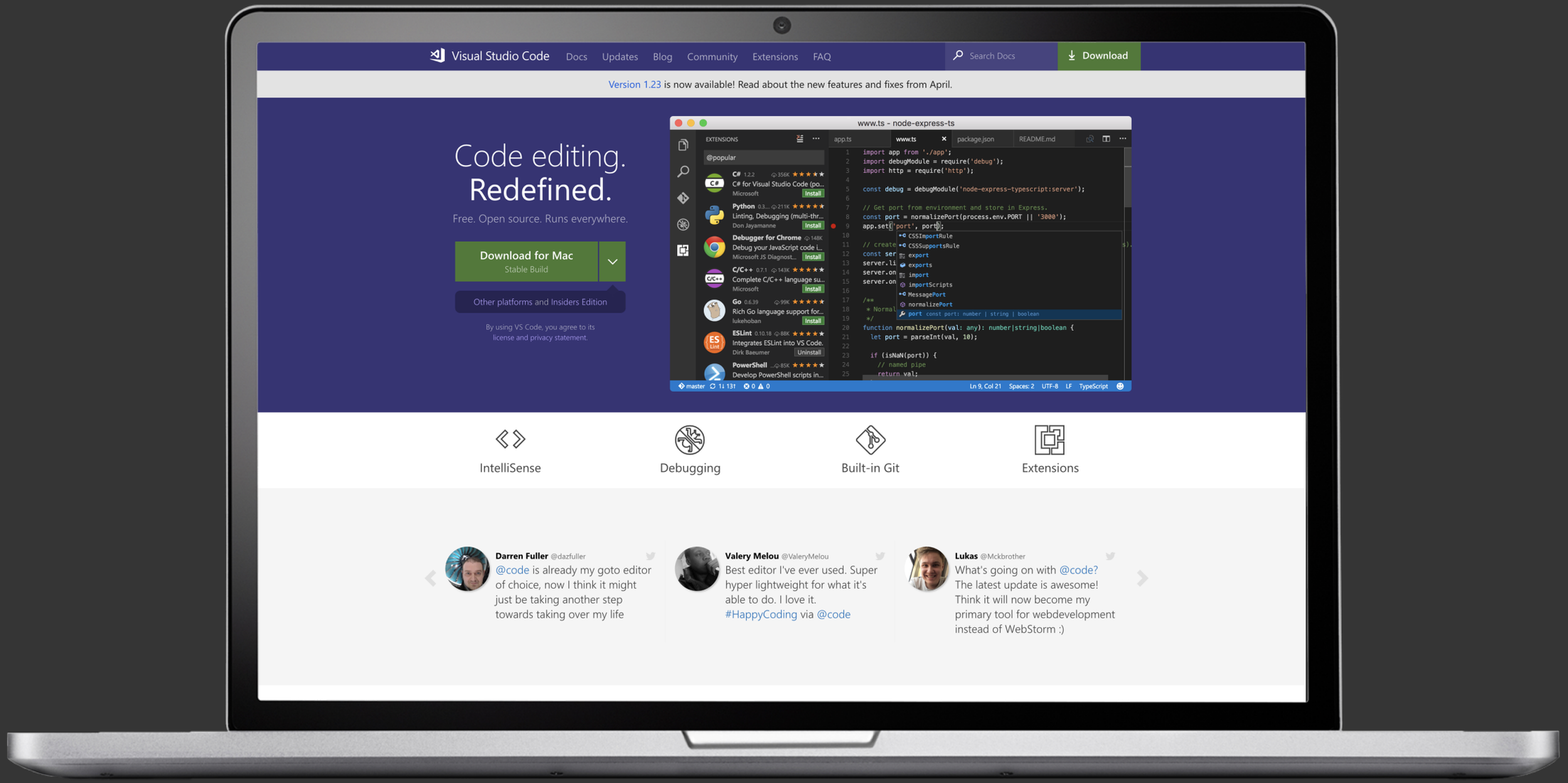
Azure for Java Developers - http://java.ms



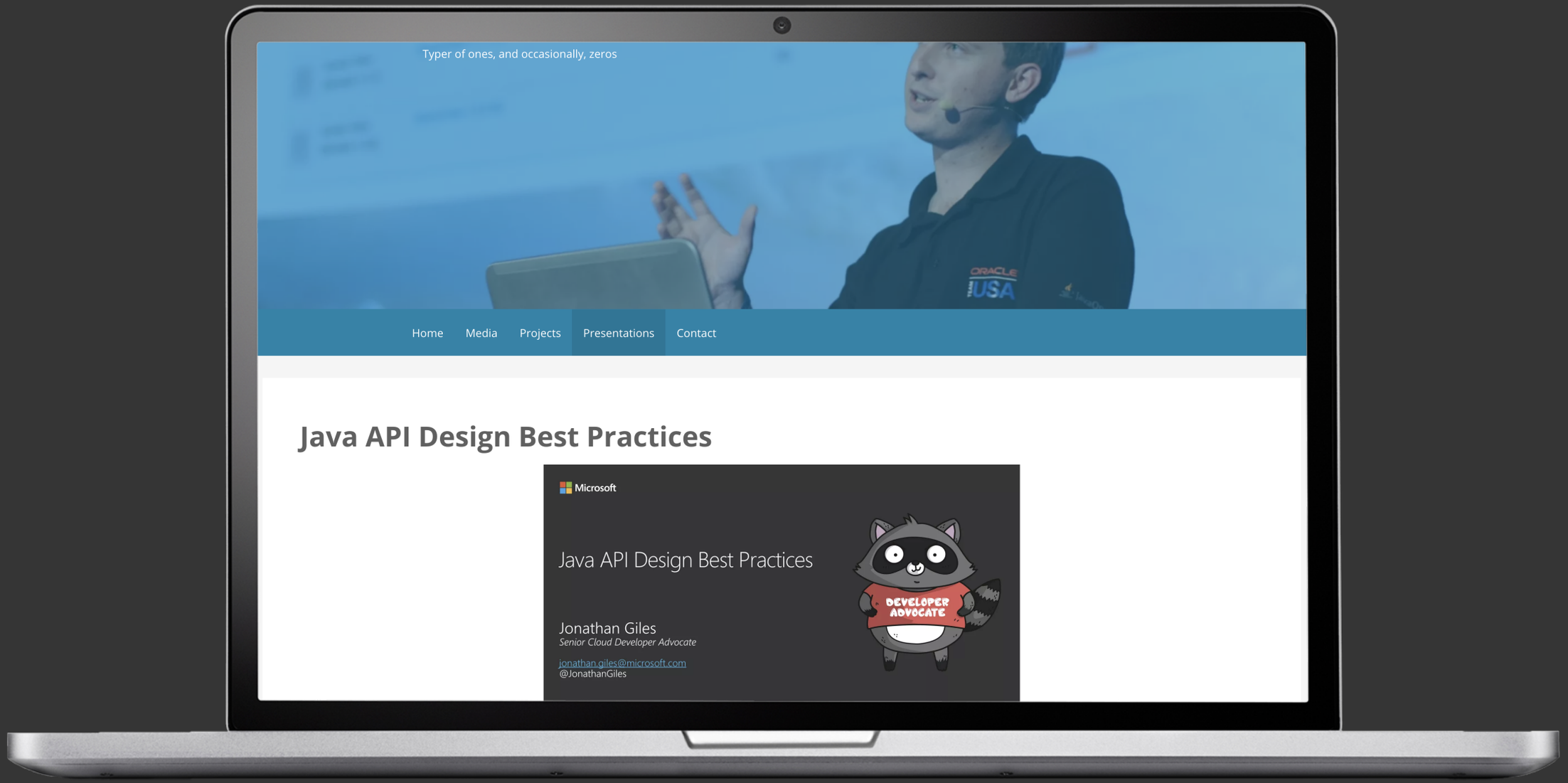
Free Azure Tier - http://java.ms/free



VS Code- http://java.ms/vscode



Presentation Materials - http://jogil.es/api-design



Microsoft

Thanks!

Jonathan Giles
Java Guy at Microsoft

jonathan.giles@microsoft.com
@JonathanGiles