Microsoft

Java API Design Best Practices

Jonathan Giles
Senior Cloud Developer Advocate

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

API Design Theory

What Is API Design?

What Is API Design?

- An API is what a developer uses to achieve some task
  - If API didn't exist, we'd be programming everything in 'raw' Java – no JDK, no Maven Central, etc – just the Java language
  - API abstracts implementation, allowing us to work at a high level of abstraction compared with 'raw' Java

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

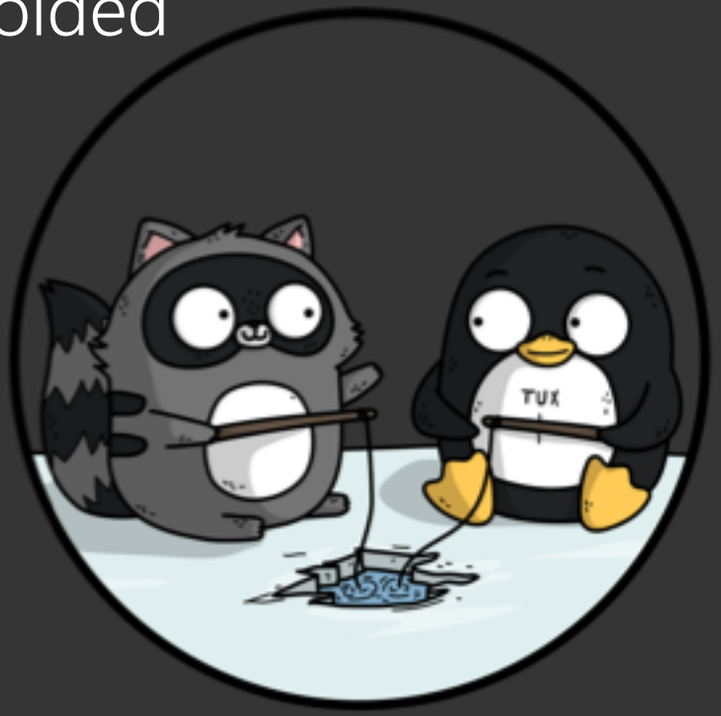
We are all API Designers

API Characteristics

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

API is a Contract

- API has to be thought of as a **contract**
  - Adding new API is acceptable
  - Removing or modifying existing API should be avoided



The Journey to 1.0.0

- API design is cheap
  - Spend cycles on it before committing to implementation

Justify Everything

- 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?"

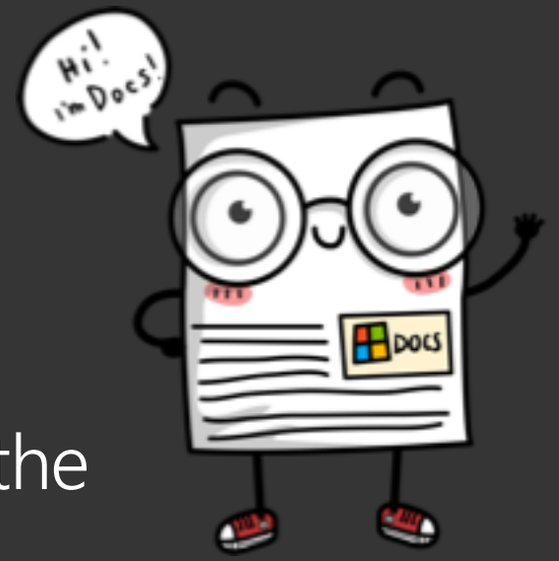
Consistency

- Consistency enables developers to intuit new API
- Important considerations include
  - Return types, e.g. List / Collection / Iterator / Iterable / Stream
  - Method naming patterns
  - Argument order
  - Consistent instantiation process

Developer Empathy & Gut Feeling

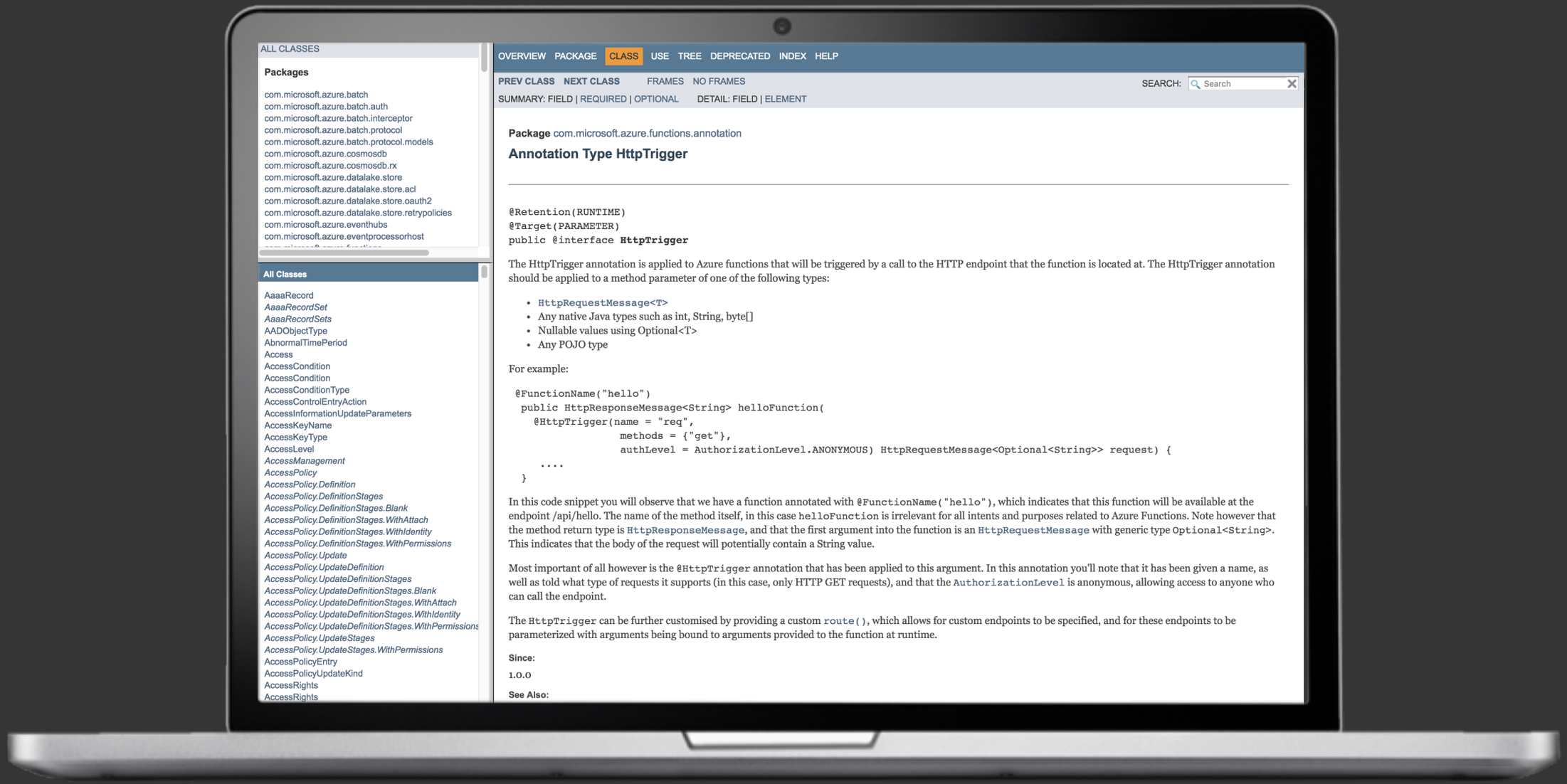
- See the problem domain from your users eyes
  - Write sample code with your API and discuss it with real users
  - Review sample code for
    - Unclear intentions
    - Duplicate, or redundant code
    - Abstraction is too low-level or too high-level
- Ultimately, a good API design comes from practice
  - Find a mentor who will provide quality feedback

Documentation



- Write quality JavaDoc
  - Include small code snippets demonstrating how to use the class
  - Make use of common ‘annotations’ to help readers (@see, @since, @link, etc).

Documentation



ALL CLASSES

Packages

com.microsoft.azure.batch
com.microsoft.azure.batch.auth
com.microsoft.azure.batch.interceptor
com.microsoft.azure.batch.protocol
com.microsoft.azure.batch.protocol.models
com.microsoft.azure.cosmosdb
com.microsoft.azure.cosmosdb.rx
com.microsoft.azure.datalake.store
com.microsoft.azure.datalake.store.acl
com.microsoft.azure.datalake.store.oauth2
com.microsoft.azure.datalake.store.retrypolicies
com.microsoft.azure.eventhubs
com.microsoft.azure.eventprocessorhost

All Classes

AaaaRecord
AaaaRecordSet
AaaaRecordSets
AADObjectType
AbnormalTimePeriod
Access
AccessCondition
AccessCondition
AccessConditionType
AccessControlEntryAction
AccessInformationUpdateParameters
AccessKeyName
AccessKeyType
AccessLevel
AccessManagement
AccessPolicy
AccessPolicy.Definition
AccessPolicy.DefinitionStages
AccessPolicy.DefinitionStages.Blank
AccessPolicy.DefinitionStages.WithAttach
AccessPolicy.DefinitionStages.WithIdentity
AccessPolicy.DefinitionStages.WithPermissions
AccessPolicy.Update
AccessPolicy.UpdateDefinition
AccessPolicy.UpdateDefinitionStages
AccessPolicy.UpdateDefinitionStages.Blank
AccessPolicy.UpdateDefinitionStages.WithAttach
AccessPolicy.UpdateDefinitionStages.WithIdentity
AccessPolicy.UpdateDefinitionStages.WithPermissions
AccessPolicy.UpdateStages
AccessPolicy.UpdateStages.WithPermissions
AccessPolicyEntry
AccessPolicyUpdateKind
AccessRights
AccessRights

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES SEARCH: Search

SUMMARY: FIELD | REQUIRED | OPTIONAL DETAIL: FIELD | ELEMENT

Package com.microsoft.azure.functions.annotation

Annotation Type HttpTrigger

```
@Retention(RUNTIME)
@Target(PARAMETER)
public @interface HttpTrigger
```

The HttpTrigger annotation is applied to Azure functions that will be triggered by a call to the HTTP endpoint that the function is located at. The HttpTrigger annotation should be applied to a method parameter of one of the following types:

- `HttpRequestMessage<T>`
- Any native Java types such as int, String, byte[]
- Nullable values using Optional<T>
- Any POJO type

For example:

```
@FunctionName("hello")
  public HttpResponseMessage<String> helloFunction(
    @HttpTrigger(name = "req",
                 methods = {"get"},
                 authLevel = AuthorizationLevel.ANONYMOUS) HttpRequestMessage<Optional<String>> request) {
      ....
  }
```

In this code snippet you will observe that we have a function annotated with `@FunctionName("hello")`, which indicates that this function will be available at the endpoint /api/hello. The name of the method itself, in this case `helloFunction` is irrelevant for all intents and purposes related to Azure Functions. Note however that the method return type is `HttpResponseMessage`, and that the first argument into the function is an `HttpRequestMessage` with generic type `Optional<String>`. This indicates that the body of the request will potentially contain a String value.

Most important of all however is the `@HttpTrigger` annotation that has been applied to this argument. In this annotation you'll note that it has been given a name, as well as told what type of requests it supports (in this case, only HTTP GET requests), and that the `AuthorizationLevel` is anonymous, allowing access to anyone who can call the endpoint.

The `HttpTrigger` can be further customised by providing a custom `route()`, which allows for custom endpoints to be specified, and for these endpoints to be parameterized with arguments being bound to arguments provided to the function at runtime.

Since:

1.0.0

See Also:

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.

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.

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

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

Generated APIs

- Today, many APIs are auto-generated from, e.g. Swagger specs
  - This is because a lot of API wraps web services
  - Microsoft uses this for many Azure SDKs
  - Tools like the free AutoRest from Microsoft do this
  - Regardless of tool, it normally leads to a robotic API

Generated APIs

- If you need to generate APIs like this:
  - Evaluate tooling options – some produce better output than others
  - Evaluate very carefully the output – review JavaDoc!
- Convenience layer enables best of both worlds:
  - Rapid development and iteration of auto-gen APIs (from changes in spec)
  - Ability to add cleaner, more developer-friendly layer on top
- Both layers could be released to Maven Central, developers can then choose what works best for them

In conclusion:

There is no magical process to API design.

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

Practical Advice

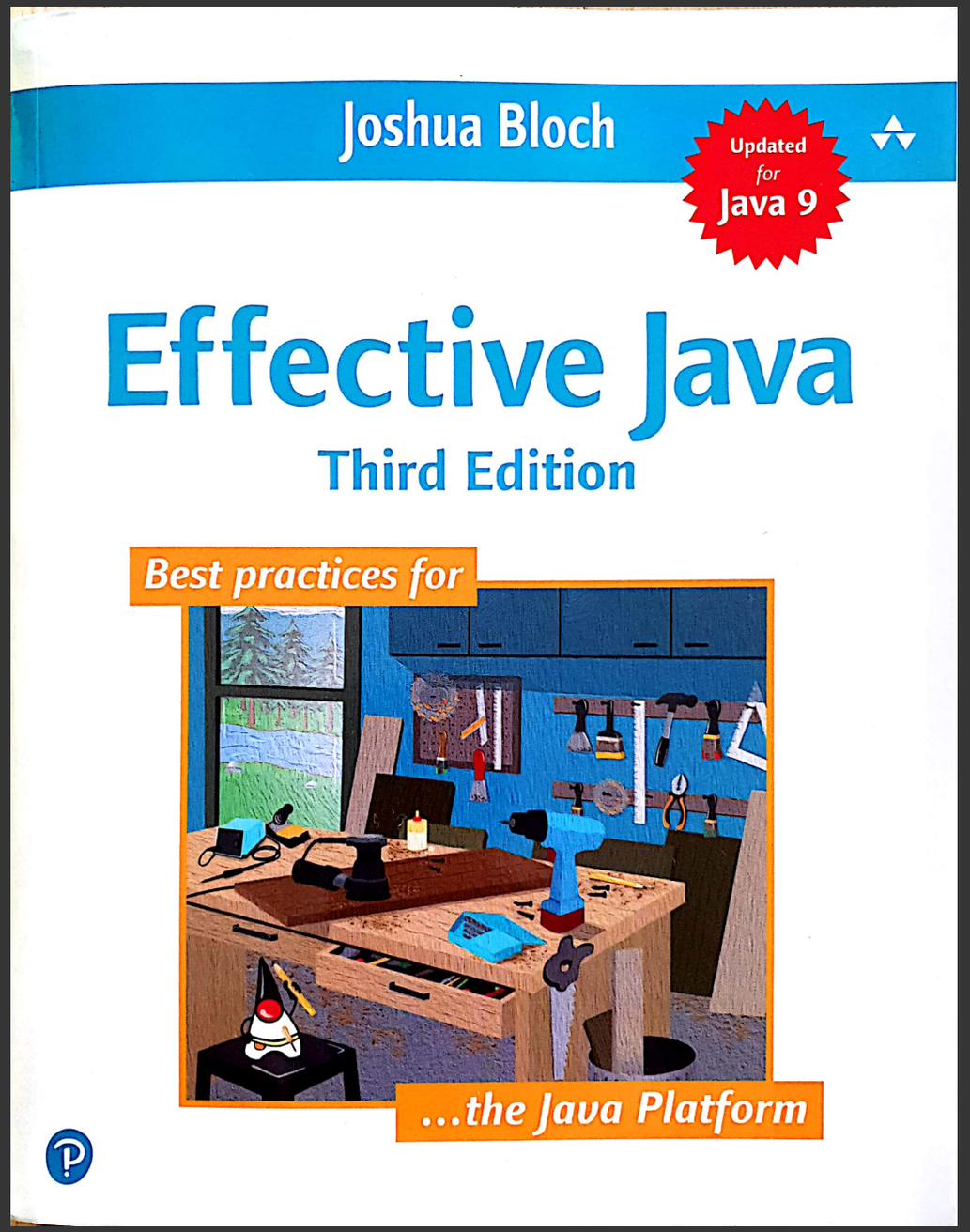
Effective Java 3rd Edition

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.



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://ilordiales.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://ilordiales.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: Minimise API

- Public API is a contract
  - 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
- Start with private modifiers, and increase visibility only after consideration
  - Fields should rarely be public

Tip 2: Minimise API

- Understand, and properly manage, implementation classes
  - Doing this makes it easier to modularize under JDK 9

- 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!

Tip 3: Intentional Inheritance

- Our default position should be to make classes and public methods final
  - This, again, helps to reduce our API surface area
- Introduce protected API carefully
  - Before committing to it, write subclasses that use it

Tip 4: Minimise Exposing External Dependencies

- 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
- Fashions can change faster than your API can!
  - Choose which horse to bet on – don't bet on all of them!
    - e.g. Offer one async approach (e.g. RxJava), not multiple approaches

Tip 5: Don't Return null

- 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 (but refer to next tip)

Tip 6: 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 6: 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 6: 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 6: 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;
  }
  ...
}
```

'Effective Java' Item 55

Tip 6: 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.

'Effective Java' Item 61

Tip 7: Beware of Boxing

- Boxing / Unboxing is when Java converts primitives to / from reference types
  - e.g. int <-> Integer
- There are three concerns when using reference types:
  - Increased possibility of null pointer exceptions (no possibility with primitives)
  - Correctness (e.g. == operator works differently for int and Integer)
  - Performance considerations

Tip 7: Beware of Boxing

- Correctness:
  - Reference types are objects, so == compares object identity.
  - This may not be what is intended!

Tip 7: Beware of Boxing

```
// Two int primitive types, both with same value
int value2Primitive = 2;
int value2PrimitiveAgain = 2;

// Testing equality using == operator, returns true – as expected
System.out.println(value2Primitive == value2PrimitiveAgain);

// Two Integer reference types, both representing the same value of 2
Integer value2 = new Integer(2);
Integer value2Again = new Integer(2);

// What do we get here?
System.out.println(value2.equals(value2Again));
      // true – as expected
System.out.println(value2 == value2Again);
      // false – might be surprising!
```

'Effective Java' Item 61

Tip 7: Beware of Boxing

- Performance:
  - Unless there is a reason to use the reference type, use primitives by default
  - Boxing / Unboxing can sap performance in some API

Tip 7: Beware of Boxing

```
// Simple code snippet, note the use of Long to accumulate the sum value
long t = System.currentTimeMillis();
Long sum = 0L;
for (long i = 0; i < Integer.MAX_VALUE; i++) {
  sum += i;
}

System.out.println("total: " + sum);
System.out.println("processing time: " + (System.currentTimeMillis() - t) + " ms");

// total: 2305843005992468481
// processing time: 6756 ms
```

https://dzone.com/articles/java-performance-notes-autoboxing-unboxing

Tip 7: Beware of Boxing

```
// Same code snippet, with a primitive long to accumulate the sum value
long t = System.currentTimeMillis();
long sum = 0L;
for (long i = 0; i < Integer.MAX_VALUE; i++) {
  sum += i;
}

System.out.println("total: " + sum);
System.out.println("processing time: " + (System.currentTimeMillis() - t) + " ms");

// total: 2305843005992468481
// processing time: 1248 ms

// In summary: the reference type approach is more than 5x slower, for no benefit!
```

https://dzone.com/articles/java-performance-notes-autoboxing-unboxing

Tip 7: Beware of Boxing

- In summary:
  - When you see a return type or method parameter using reference types, e.g. Integer, consider it a code smell
  - In most circumstances this may not be too bad – coders will probably deal with it fine
    - But, it is a location where there is a non-zero chance of issues, all of which would be negated if primitive types were used in their place
    - If the code is to be used in tight, performance-critical loops, the cost of autoboxing can be extreme

It is not appropriate to use autoboxing and unboxing for scientific computing, or other performance-sensitive numerical code. An Integer is not a substitute for an int; autoboxing and unboxing blur the distinction between primitive types and reference types, but they do not eliminate it.

https://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html

Tip 8: 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 8: 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 8: Become Familiar With java.util.function

Interface	Signature	Example
UnaryOperator<T>	T apply(T t)	`List<String> names = Arrays.asList("bob", "josh", "megan");` `names.replaceAll(String::toUpperCase);`

BinaryOperator<T>	T apply(T t1, T t2)	`Map<String, Integer> salaries = new HashMap<>();`
		`salaries.put("John", 40000);`
		
		`salaries.replaceAll((name, oldValue) ->`
		`  name.equals("Freddy") ? oldValue : oldValue + 10000);`

Predicate<T>	boolean test(T t)	`List<String> namesWithA = names.stream()`
		`  .filter(name -> name.startsWith("A"))`
		`  .collect(Collectors.toList());`

Function<T,R>	R apply(T t)	`Map<String, Integer> nameMap = new HashMap<>();` `Integer value = nameMap.computeIfAbsent("Giles", String::length);`

Supplier<T>	T get()	`int[] fibs = {0, 1};`
		`Stream<Integer> fibonacci = Stream.generate(() -> {`
		`  int result = fibs[1];`
		`  int fib3 = fibs[0] + fibs[1];`
		`  fibs[0] = fibs[1];`
		`  fibs[1] = fib3;`
		`  return result;`
		`});`

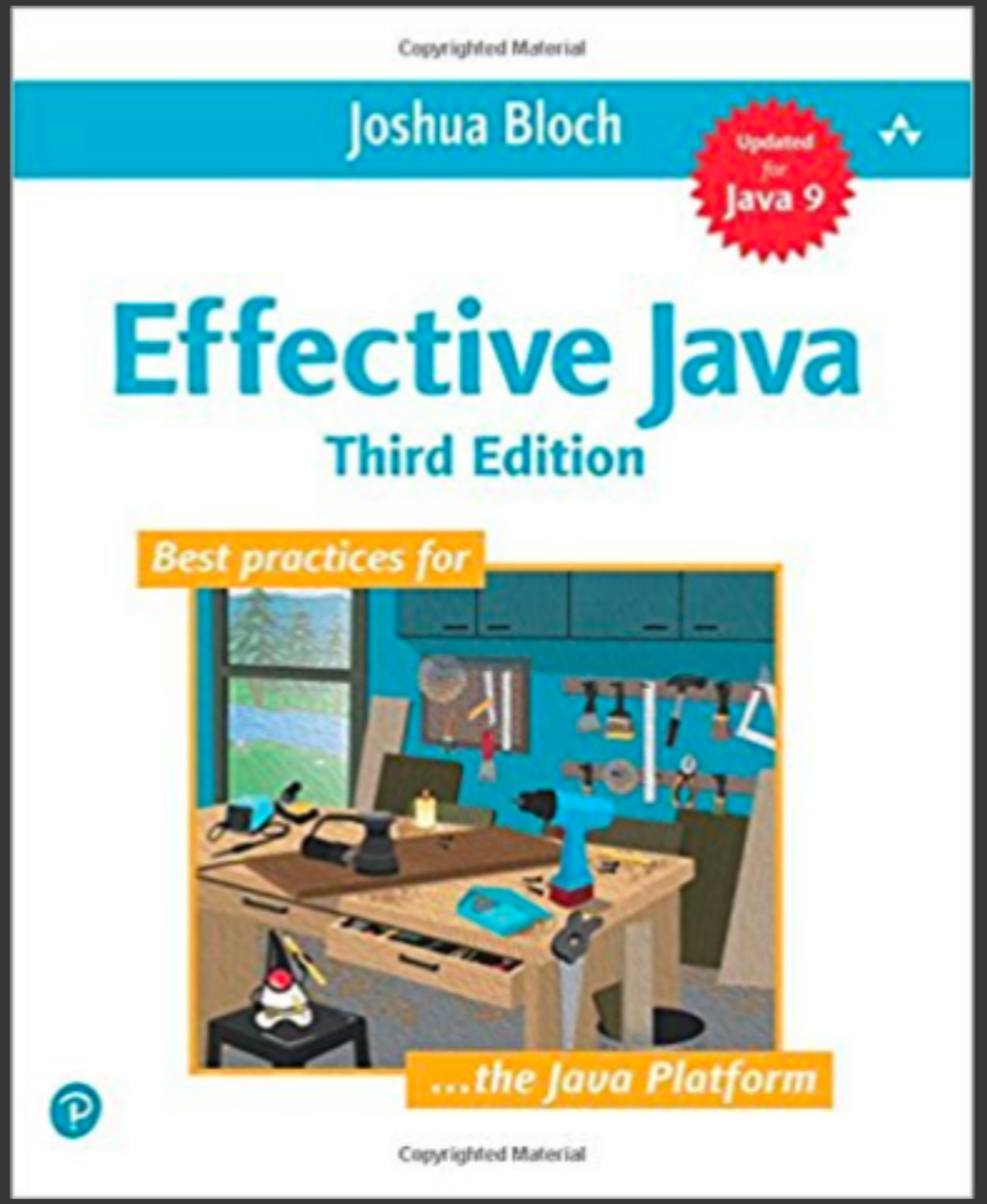
Consumer<T>	void accept(T t)	`List<String> names = Arrays.asList("John", "Freddy", "Samuel");` `names.forEach(name -> System.out.println("Hello, " + name));`

http://www.baeldung.com/java-8-functional-interfaces

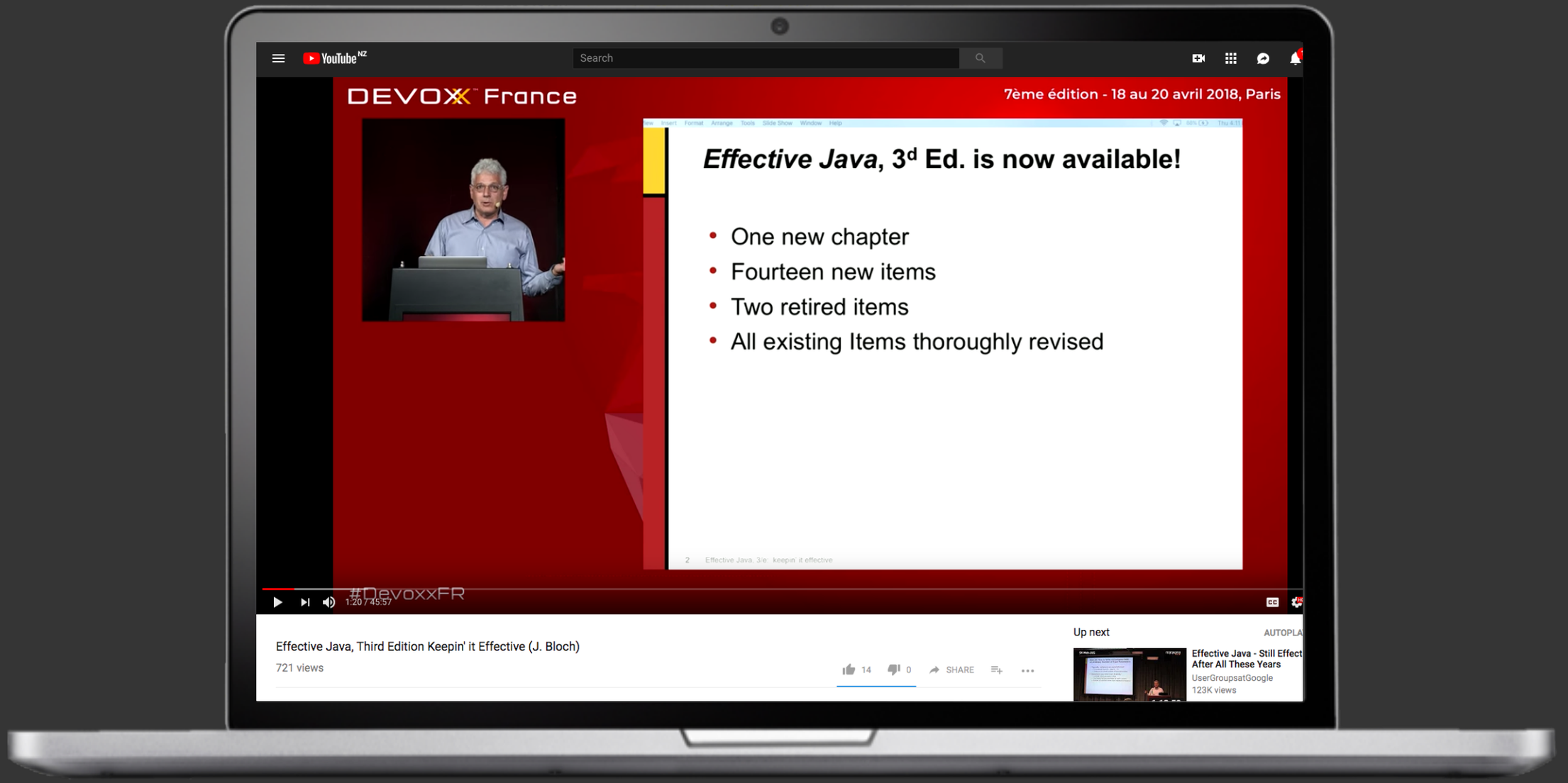
Tip 8: Become Familiar With java.util.function

- 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

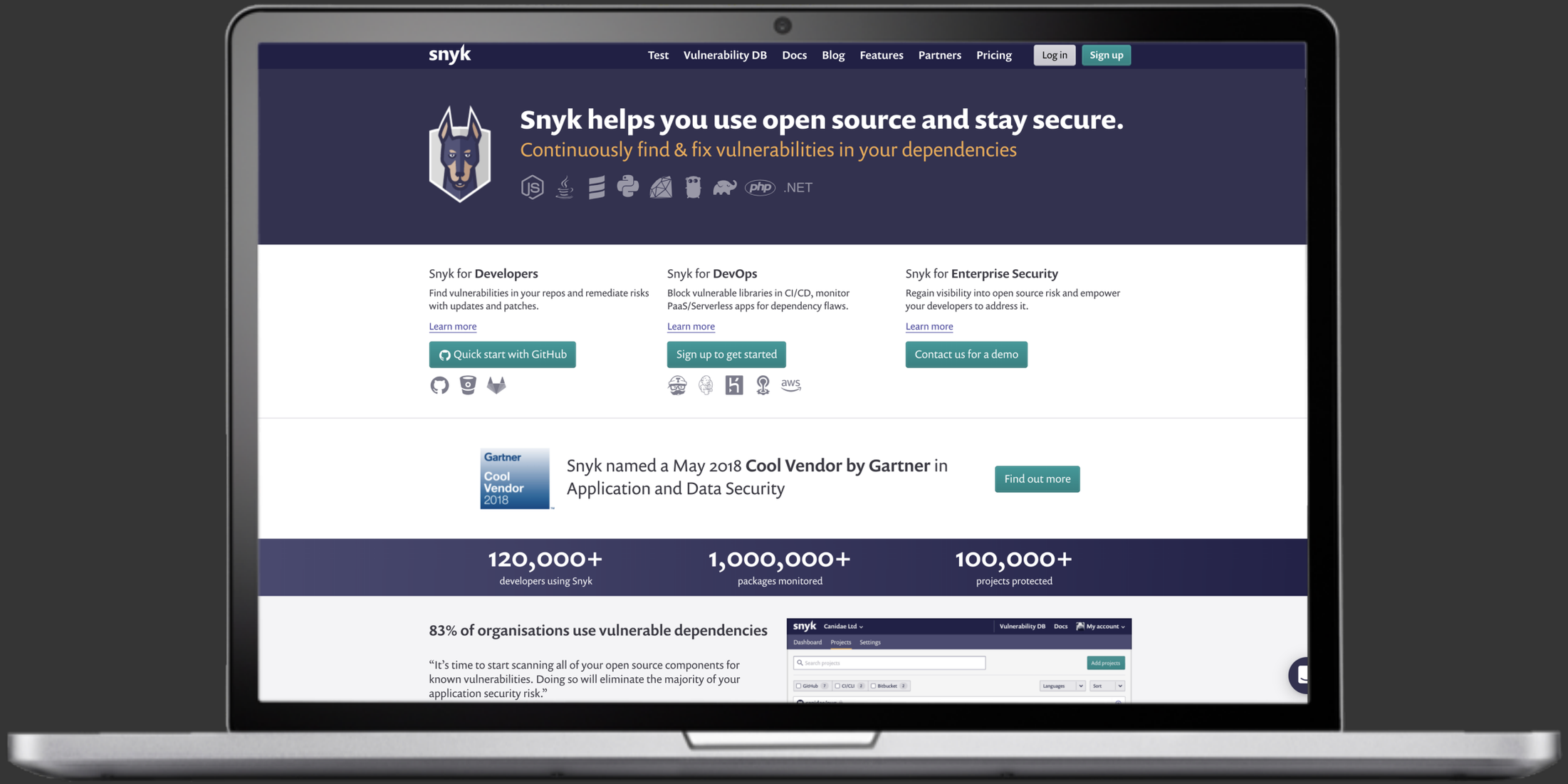
Resources


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

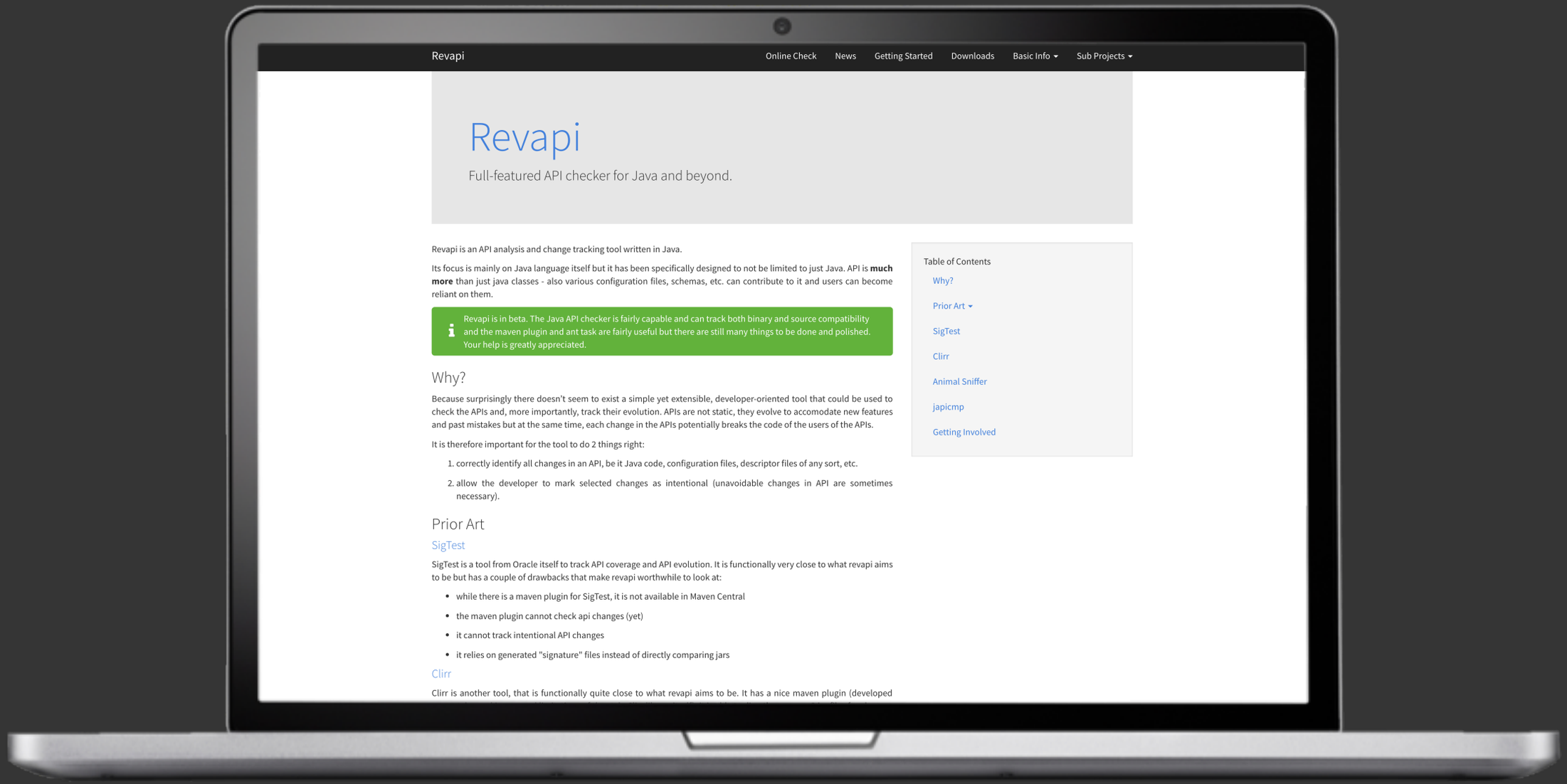
YouTube


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
AUTOPLA
Effective Java - Still Effect
After All These Years
UserGroupsatGoogle
123K views

Snyk – http://snyk.io

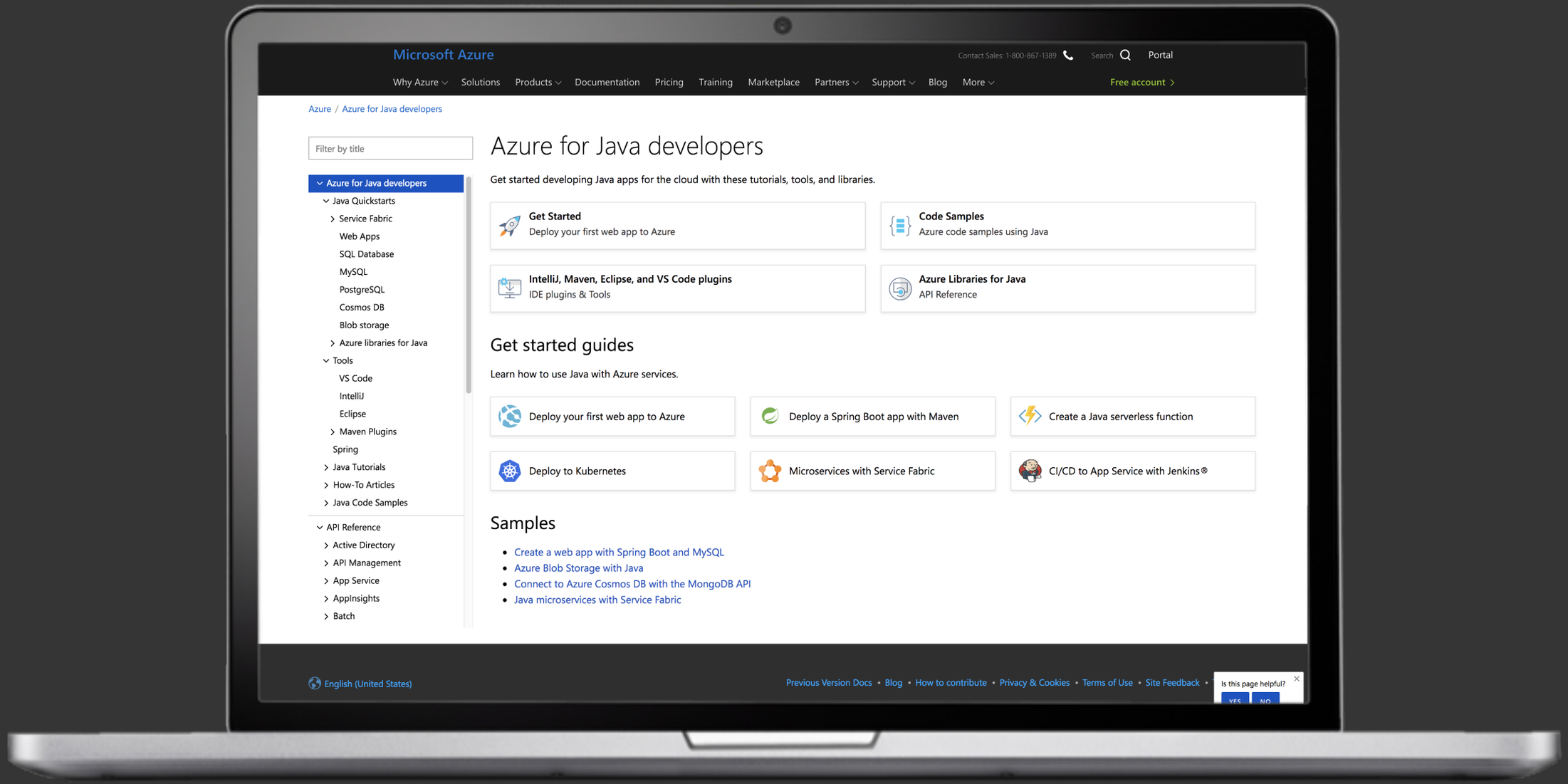


RevAPI – http://revapi.org

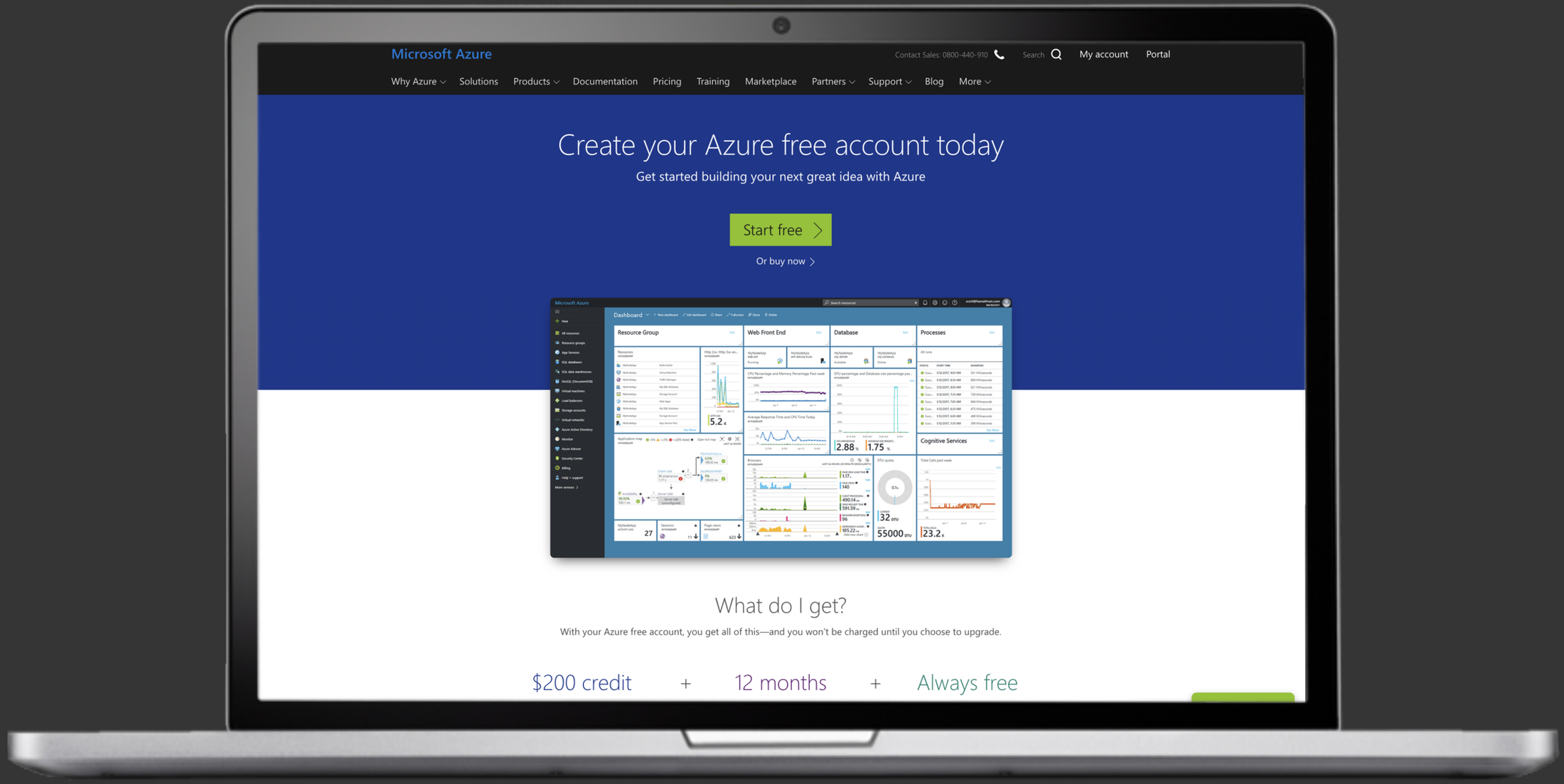


Call To Action

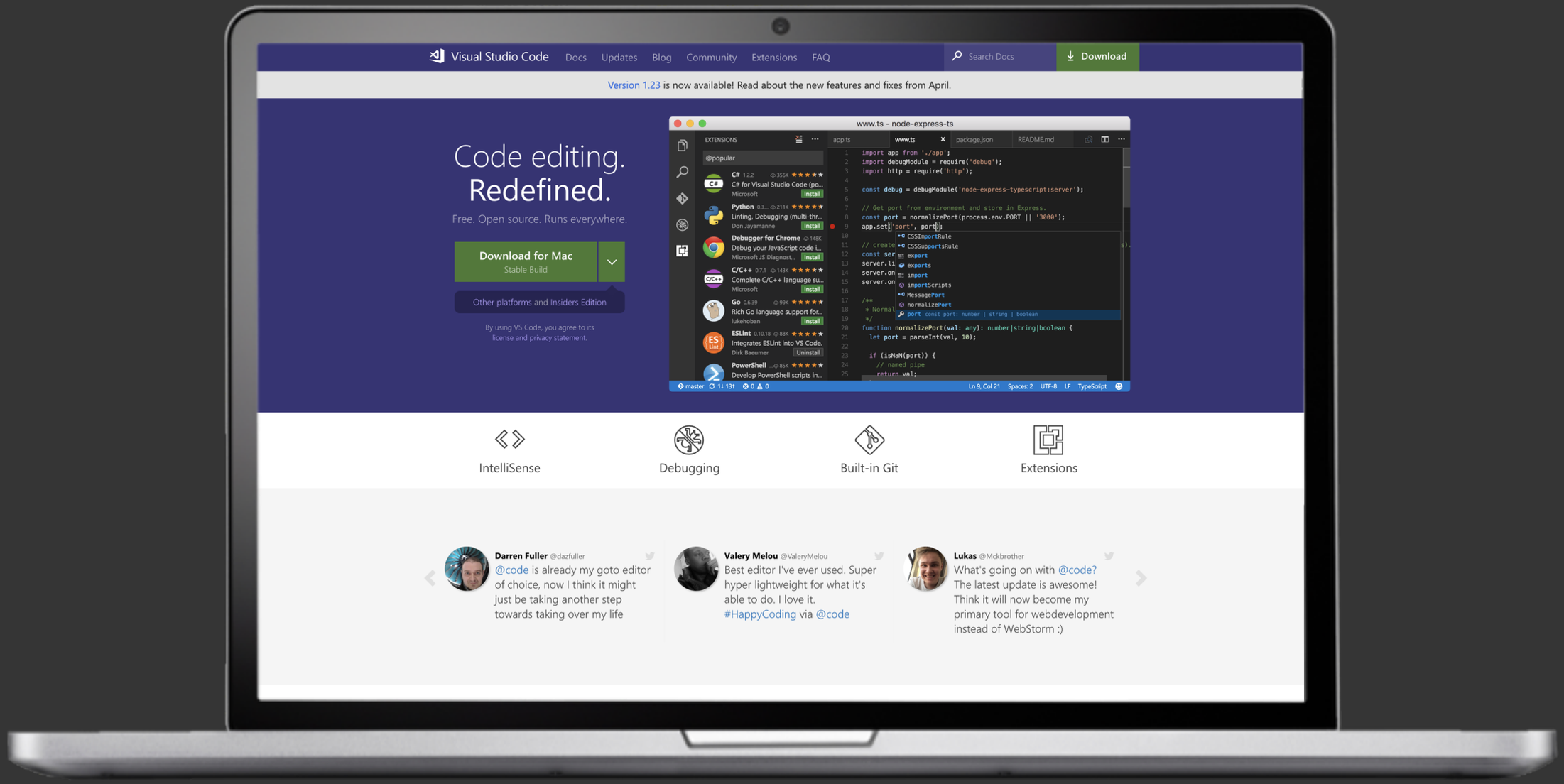
Azure for Java Developers - http://java.ms



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



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



Microsoft

Thanks!

Jonathan Giles
Senior Cloud Developer Advocate

jonathan.giles@microsoft.com
@JonathanGiles

More Tips

Warning!

These are ‘overflow’ slides. They exist because they represent important concepts that I might cover if time permits, but they are not as fully polished as the main deck. Proceed with caution!

Tip: Builders > Constructor Telescoping

- Constructor arguments are important
  - Used to specify subset of parameters required to form a correct instance
  - Sometimes there are varying options – constructors can `telescope`
  - Argument overload – hard to write and hard to read
- Builders are a better option in cases where there are many constructors or constructor arguments

Tip: Builders > Constructor Telescoping

```
public class Pizza {
  public Pizza(int size) { … }
  public Pizza(int size, boolean cheese) { … }
  public Pizza(int size, boolean cheese, boolean pepperoni) { … }
  public Pizza(int size, boolean cheese, boolean pepperoni, boolean bacon) { … }
  …
}
```

http://java.ms/api-builder-pattern

Tip: Builders > Constructor Telescoping

An alternative approach is to use the JavaBean pattern with setters:

```
public class Pizza {
  public Pizza(int size) { }
  public void setCheese(boolean val) { … }
  public void setPepperoni(boolean val) { … }
  public void setBacon(boolean val) { … }
  …
}
```

http://java.ms/api-builder-pattern

Tip: Builders > Constructor Telescoping

We could also use static factory methods from tip 1, but difficult to determine a useful API, given the possible pizza permutations

```
public class Pizza {
  private Pizza() { }
  public static Pizza createSmallCheesePepperoniPizza() { … }
  public static Pizza createMediumCheeseBaconPepperoniPizza() { … }

  // Impossible to support all permutations!
}
```

http://java.ms/api-builder-pattern

Tip: Builders > Constructor Telescoping

```
public class Pizza {
  private final int size;
  private final boolean cheese;
  private final boolean pepperoni;
  private final boolean bacon;

  // Builder class here – on next slide

  private Pizza(Builder builder) {
    size = builder.size;
    cheese = builder.cheese;
    pepperoni = builder.pepperoni;
    bacon = builder.bacon;
  }
}
```

http://java.ms/api-builder-pattern

Tip: Builders > Constructor Telescoping

```
public static class Builder {
  //required
  private final int size;

  //optional
  private boolean cheese;
  private boolean pepperoni;
  private boolean bacon;

  public Builder(int size) {
    this.size = size;
  }

  public Builder addCheese() {
    cheese = true;
    return this;
  }

  public Builder addPepperoni() {
    pepperoni = true;
    return this;
  }

  public Builder addBacon() {
    bacon = true;
    return this;
  }

  public Pizza build() {
    return new Pizza(this);
  }
}
```

http://java.ms/api-builder-pattern

Tip: Builders > Constructor Telescoping

```
VirtualMachine linuxVM = azure.virtualMachines().define("myLinuxVM")
  .withRegion(Region.US_EAST)
  .withNewResourceGroup(rgName)
  .withNewPrimaryNetwork("10.0.0.0/28")
  .withPrimaryPrivateIPAddressDynamic()
  .withNewPrimaryPublicIPAddress("mylinuxvmdns")
  .withPopularLinuxImage(UBUNTU_SERVER_16_04_LTS)
  .withRootUsername("tirekicker")
  .withSsh(sshKey)
  .withSize(VirtualMachineSizeTypes.STANDARD_D3_V2)
  .create();
```

Tip: 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: 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: Support Lambdas

Java Swing UI Toolkit:

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

Tip: 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));
```

Tip: Don't Use Stream as a Return Type

- APIs need to return collections, and there are many options
  - Collection, Set, List, Iterable, arrays, and now Streams!
- Bad options:
  - Returning a Stream is problematic:
    - Difficult to do a normal *for* loop (without additional code)
  - Returning Iterable makes stream processing difficult
    - Need to wrap the Iterable with additional code to create a Stream

Tip: Don't Use Stream as a Return Type

- Best options:
  - Collection (and subtypes) support external (*for-each*) and internal (streams) iteration.
  - Arrays are valid too, as Arrays.asList() and Stream.of() methods enable quick conversion

Tip: Interfaces > Abstract Classes

- It is much easier to retrofit a class to implement a new interface, rather than extend a new abstract class
- From Java 8, interfaces now support default methods
  - Many of the benefits of abstract classes

Tip: Maximise Immutability

- Immutable classes are simple
  - Once instantiated they do not change
  - Thread-safe, testable, and easily reasoned about
- Mutable classes increase code complexity
- Strive to make fields `final` whenever possible

Tip: Avoid Raw Types

- Generics are great – we should always use them
  - A raw type is when a generic class is used without generic typing, e.g. List
  - We do this to ensure type safety and expressiveness
  - Raw types exist so that generics, when introduced in Java 5, were backwards compatible
- If your IDE is warning you about a raw type, add generic type information!

Tip: Prevent Utility Class Instantiation

- Utility classes are pervasive in public API
  - Often a collection of static methods
  - The intent of utility classes is to simply call these methods
- If no constructor is defined in a class, a default, public constructor is provided
  - There is no use in having a constructor on a pure utility class!
  - Utility classes should have a private constructor to prevent instantiation

Tip: Prevent Utility Class Instantiation

```
public class Utils {
  private Utils() {
    // prevent instantiation of class
  }

  public static void sum(int v1, int v2) { … }
  …
}
```

Tip: Favour Enums Over Boolean Arguments

- Be wary of API with boolean parameters
  - Re-reading code with boolean parameters can be difficult
- If the boolean parameter is used in multiple methods, consider introducing an enum

'Effective Java' Item 29, 30

Tip: Use Generics

- Generics have been around since Java 5
  - We should all be writing generic classes by now
  - Generics help code quality – it's harder to do something dumb…
- Sometimes generics can create a lot of noise, e.g.

Tip: Use Generics

Sometimes generics can create a lot of noise, e.g.

```
public Single<RestResponse<MyGetHeaders, Flowable<ByteBuffer>>> getWithRestResponseAsync()
```

In some cases an intermediate class aids readability:

```
public final class MyGetResponse extends RestResponse<MyGetHeaders, Flowable<ByteBuffer>> {
  MyGetResponse(int statusCode,
                MyGetHeaders headers,
                Map<String, String> rawHeaders,
                Flowable<ByteBuffer> body) {
    super(...);
  }
}
```

As we can now write this:

```
public Single<MyGetResponse> getWithRestResponseAsync()
```

Tip: Avoid Unchecked Warnings

- Generics give the compiler useful information
  - This enables more checks to be run at compile time
  - Generics lead to more 'unchecked' warnings
  - Do not ignore the increased verbosity – read and consider what it means
- Eliminating unchecked warnings improves type safety
  - When you can't eliminate a warning, but can prove type safety manually, use the @SuppressWarnings("unchecked") annotation

Tip: Avoid Unchecked Warnings

```
// A method that takes a raw List and casts it to a List<String>
void foo(List inputList) {
  List<String> list = (List<String>) inputList; // unsafe cast
}
```

Tip: Avoid Unchecked Warnings

```
// A method that takes a raw List and casts it to a List<String>
@SuppressWarnings("unchecked")
void foo(List inputList) {
  List<String> list = (List<String>) inputList; // unsafe cast
}
```

Tip: Avoid Unchecked Warnings

```
// A method that takes a raw List and casts it to a List<String>
void foo(List inputList) {
  // We can suppress this because it is only ever called by foo(List<String>)
  @SuppressWarnings("unchecked")
  List<String> list = (List<String>) inputList; // unsafe cast
}
```

Tip: Use Generics

- While everyone (hopefully!) knows and uses class generics, less people use generic methods.
  - We've already encountered them, e.g the new static List.of() method
- Generic methods are methods that introduce their own type parameters.
  - Similar to declaring a generic class, but the type parameter's scope is limited to the method where it is declared.
  - Static and non-static generic methods are allowed

Tip: Use Generics

Static generic method example:

```
public class Util {
  public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
    return p1.getKey().equals(p2.getKey()) && p1.getValue().equals(p2.getValue());
  }
}
```

Tip: Builders > JavaBean Setters

- Constructors ensure a valid initial state
- Simplifying constructors by removing arguments and replacing with setters is a bad idea
  - Calling setters is not enforced
  - Developers may ignore required properties and leave class instance in an invalid state