



Clean Code



Uncle Bob

Robert Cecil Martin

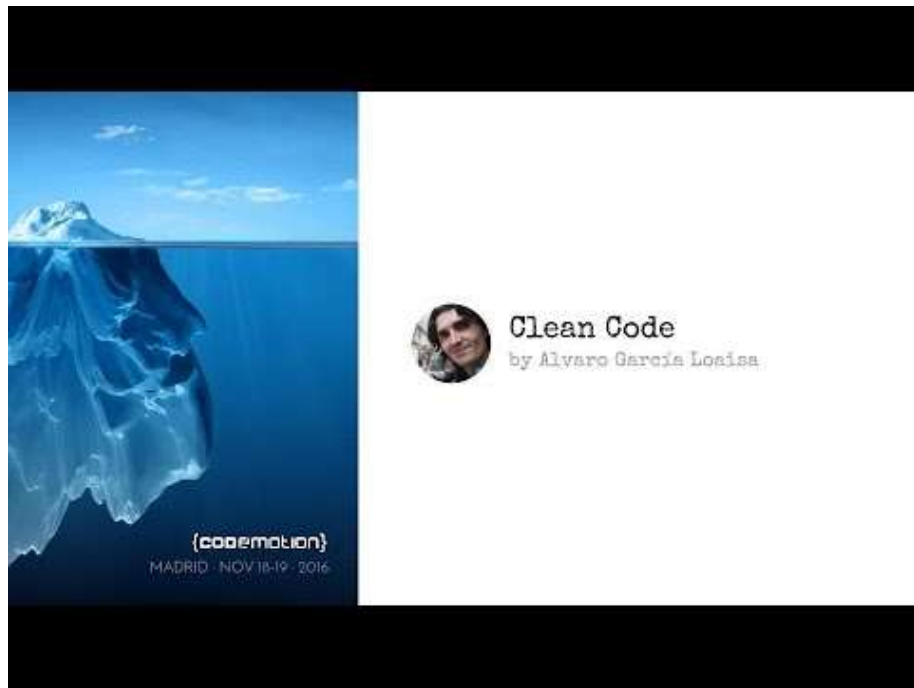
@unclebobmartin

<http://blog.cleancoder.com/>

Agile Manifesto, Extreme
Programming, UML for Java...
Clean Coders

Advertencias

- ▶ Lo que vamos a ver son recomendaciones. No son la biblia. Cada caso es diferente y el sentido común está por encima de las normas.
- ▶ Hay cientos (¿miles?) de videos y slides sobre el tema
<https://goo.gl/r25If>
- ▶ Es el resumen de un libro



Advertencias

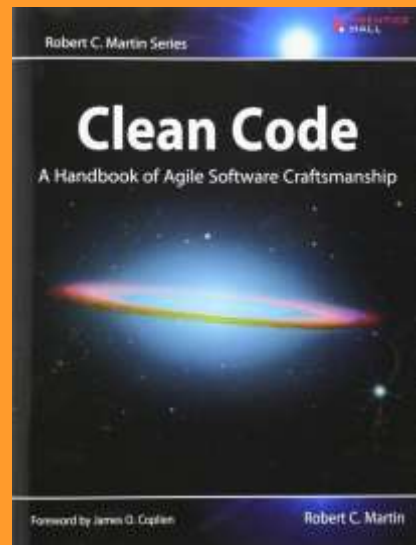
- ▶ Quien esté libre de pecado....



Clean Code

A Handbook of Agile Software Craftsmanship

- ❑ Publicado en 2008
- ❑ Resumen de buenas prácticas ya conocidas e intuitas pero no siempre llevadas a la práctica
- ❑ El código limpio no es solo algo deseable. Es algo vital para compañías y programadores.
- ❑ Quien escribe código *sucio* hace perder tiempo y dinero al resto intentando comprenderlo. Incluso a si mismo.



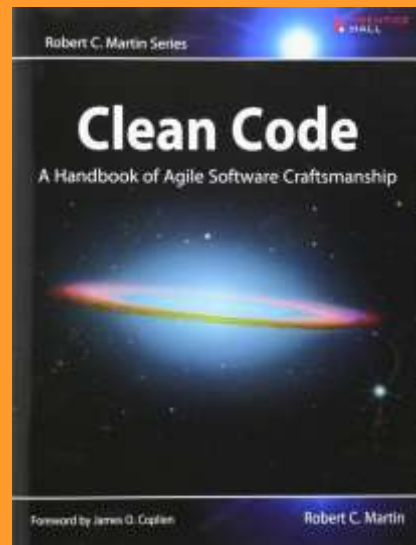
Clean Code

- ❑ Tenemos el libro en papel y en castellano
- ❑ En eBook en inglés



Yeray Darias, Codemotion 2016:

«Hay dos cosas que la humanidad no debería haber hecho: La bomba H y la traducción de Clean Code»



- ❑ Se proponen una serie de guías y buenas prácticas a la hora de escribir código
- ❑ Ejemplos en Java, pero aplicables a cualquier lenguaje de alto nivel.
- ❑ Dividido en 2 partes + 1 capítulo resumen
 - ❑ Caps. 1-13. Buenas practicas
 - ❑ Caps. 14-16. Ejemplos de situaciones reales
 - ❑ Cap. 17 . Olores y síntomas.

“

La responsabilidad de hacer buen código es de los programadores. Hay que negarse a hacer mal código.

”

Código Limpio

- ▶ El código limpio es elegante, eficaz, legible. Con pruebas unitarias que expliquen lo que hace y sirva de ejemplo
- ▶ Leemos más código del que escribimos. No podemos escribir buen código si el que está alrededor es un desastre.



La teoría de las ventanas rotas



La teoría de las ventanas rotas



Código Limpio

- ▶ Si el código es  es muy fácil 
- ▶ Si está limpio, ¿quien se atreve a poner la primera?
- ▶ Regla del Boy Scout: deja el código más limpio que como te lo encontraste.



Nombres con sentido

Nombres con sentido

- ▶ Todos los nombres tienen que ser descriptivos, **buscables** y **pronunciables**. Gasta un par de minutos en pensar un buen nombre. Si no te gusta, cámbialo luego.
- ▶ Evita abreviaturas, prefijos (notación húngara), palabras redundantes (the-, a-, -object, -data). Usa nombres que se puedan buscar. Evita variables de una sola letra (salvo i, j, k)
- ▶ Los nombres de clases comienzan con Mayúscula. Los objetos con minúscula.

Nombres con sentido

- ▶ Mejor usar “añadidos” en la implementación (que será privada y menos usada) que en el Interface (*IFactory*)
- ▶ Nombres de clase: intentar evitar sufijos tipo *Manager* o *Processor*. Nombres de clase no deben ser verbos, sino sustantivos
- ▶ Métodos: **si** deberían ser verbos
- ▶ Usa *get-set* y *is* para acceso a variables
- ▶ No uses juegos de palabras (que otros, tal vez en gitHub y otras culturas) no entenderán ni bromas.

Nombres con sentido

- ▶ Usa las mismas palabras para lo mismo. Al menos por proyecto (¿get? ¿fetch? ¿retrieve?).
- ▶ No usar la misma palabra para cosas distintas (¿add se usa para sumar o para insertar?)
- ▶ Usa métodos estáticos con el nombre del tipo de argumento esperado en lugar de múltiples constructores:

Nombres con sentido

```
new MiObjeto("22")  
new MiObjeto(22)  
new MiObjeto( Hashtable )  
.....
```



```
MiObjeto.FromString("22")  
MiObjeto.FromInteger( 22 )
```



Nombres con sentido

- ▶ Usa nombres técnicos cuando la intención ser técnica: Factory, List...
- ▶ Si no puedes usar un nombre técnico que entienda el siguiente programador, usa uno del *dominio* (negocio): al menos se podrá preguntar a alguien de negocio que es lo que significa.
- ▶ Los nombres, cuanto mas cortos (aunque claros y explícitos), mejor.

Nombres con sentido

- ▶ No añadir prefijos o contextos innecesarios: neg* , GDB* ... Deja que el IDE te ayude a encontrar
- ▶ Clase CaserEmailAddress. Si necesitas email en otro sitio, ¿usarás esa clase con ese nombre?

Funciones

- ▶ Dos reglas básicas:
 - ▶ Primera regla: Reducidas: ~ 20 líneas
 - ▶ Segunda regla: Más reducidas aún
- ▶ Que quepa en una pantalla (de 24 * 80)
- ▶ Que la podamos describir en una sola frase

Funciones

Un solo nivel de abstracción en cada función: no se deben mezclar cosas de alto nivel con bajo nivel .

Por ejemplo, en un parseador de html:

- ▶ Una llamada `getHtml()` que devuelve el String
- ▶ Un *parseHtml* que mezcla fuentes de datos
- ▶ Una llamada `.append("\n")`
- ▶ Abrir y cerrar Streams o ficheros de disco

Funciones – Nivel de abstracción

```
public void hacerTrabajosCaseros(){
    pasearPerro();
    sacarBasura();
    for( Pieza pieza in vajillaSucias ){
        aclaraVajilla( pieza ) ;
        meteEnLavaplatos( pieza ) ;
    }
}
```

```
public void hacerTrabajosCaseros(){
    pasearPerro();
    sacarBasura();
    lavarVajillas(vajillaSucias) ;
}
```

```
private void lavarVajillas( ArrayList<Pieza> piezas) {
    for( Pieza pieza in vajillaSucias ){
        aclaraVajilla( pieza ) ;
        meteEnLavaplatos( pieza ) ;
    }
}
```

Funciones

- ▶ Deben hacer solo una cosa, y hacerla bien. No deben tener *efectos secundarios*: hacen lo que se espera que hagan, y nada más.

Funciones

```
public boolean verificaUsuarioYPassword (String usuario, String password) {  
    boolean entra = false ;  
    if (OracleExecute( "SELECT COUNT(*) FROM USERS WHERE USER='usuario' AND PASS='password'" ) ) {  
        inicializaSesionDeUsuario() ; // Inicializamos la sesión del usuario  
        entra = true ;  
    }  
    return entra ;  
}
```

¿boolean?. ¿Como sabemos que tipo de error ha pasado?

Nivel de abstracción incorrecto: Acceso a bajo nivel a BD con verificación de datos correctos. Por no decir que estamos mandando el password... ¿en claro?

¿Oracle?. ¿Y si mañana es otra BD?.
¿Control de errores?

Este efecto secundario... ¡Solo queríamos verificar usuario y password!. ¿Y si tenemos que ir a otro sitio a validar la IP o lo que sea?

Comentario inutil. Más sobre esto después.

Funciones

El código se debe leer como si fuera prosa, de arriba a abajo, de mayor a menor abstracción:

```
public void leemosConfiguracionYDetallesDeUsuario() {  
    leemosConfiguracionDeUsuario() && leemosDetallesDeUsuario() ;  
}  
private void leemosDetallesDeUsuario() {  
    leerDetallesDeLdap() ;  
}  
private void leemosConfiguracionDeUsuario() {  
    leerConfigDeBaseDeDatos() ;  
}  
private void leerDetallesDeLdap() {  
}
```

Funciones

switch: (o multiples if-else)

- ▶ Son largos
- ▶ Seguro que hacen mas de una cosa
- ▶ Mas dificiles de mantener

Funciones

```
Integer calculaPagoAEmpleado( Empleado empleado ) {  
    if( empleado.type == Empleado.COMERCIAL_TIPO1 ) {  
        calculaPagoAComercial( int porcentajeComision ) ;  
    } else if ( empleado.type == Empleado.COMERCIAL_TIPO2 ) {  
        calculaPagoAComercial() + getComisionFija() ;  
    } else if ( empleado.type == Empleado.DIRECTIVO_ACCIONISTA ) {  
        calculaPagoADirectivo( int porcentajeDeBonus , int numeroDeAcciones ) ;  
    } else if ( empleado.type == Empleado.GERENTE ) {  
        calculaPagoAGerente( int porcentajeDeComision ) ;  
    } else if ( empleado.type == Empleado.SENORA_LIMPEZA ) {  
        calculaPagoAGerente( int metrosDeOficina ) ;  
    } else if (...) {  
    } else {  
        throw Exception("Tipo de empleado inválido")  
    }  
}
```

Problemas:

- Hace más de una cosa
- La lista de tipos podría ser ilimitada
- Incumple el principio de responsabilidad única: hay más de un motivo para cambiarla
- Incumple en principio de Abierto/Cerrado ("O" SOLID)
- Hipotéticamente, podría haber otros muchos módulos que tuvieran esta misma estructura de if/switch

Funciones

```
abstract class Empleado{           // o Interface
    getSalario()
    getDiaDePago()
    ...
}
```

```
class EmpleadosFactory {
    static Empleado creaEmpleadoPorTipo(String type) {
        switch(type) {
            case COMERCIAL: return new EmpleadoComercial();
            case DIRECTIVO: return new EmpleadoDirectivo();
            case GERENT: return new EmpleadoGerente();
        }
    }
}
```

```
Class EmpleadoComercial implements/extends Empleado {
    Public int getSalario() { // implementacion real para este tipo de usuario. Solo esto cambia si hay que cambiar la forma de cálculo
    }
}
```

Funciones - Argumentos de métodos

- ▶ No debiera haber mas de 2 parámetros. 1 mejor que 2
- ▶ 3 son muchos, y mas hay que justificarlo
- ▶ Intentar evitar (salvo que se justo lo que se espera) que los parámetros se cambien dentro de una función. Usar los valores de retorno.
- ▶ Un parámetro nunca deberia ser boolean. Si tenemos un boolean tenemos un *if* y eso quiere decir que la funcion hará al menos dos cosas: haz dos funciones!

Funciones - Argumentos de métodos

- ▶ Funciones/metodos con 2 parámetros ok si están relacionados

```
Point point = makePoint( float x, float y) {  
}
```

- ▶ ¿Se puede mejorar creando una nueva clase?

```
Circle circulo = makeCircle( float x, float y, float radio)
```

```
Circle circulo = makeCircle( Point point, float radio)
```

Funciones - Argumentos de métodos

- ▶ Si hay excepciones... lanza una exception!. (No tiene porque ser procesada inmediatamente)
- ▶ No devuelvas códigos (-1, -2 ..) de error, devuelve una exception o una clase que contenga información

Comentarios en código

- ▶ Solo debe haber comentarios cuando el código no sea capaz de expresar lo que hace.
- ▶ Comentarios aceptables
 - ▶ © y otros legales
 - ▶ Advertencias de porqué se ha tomado una decisión
 - ▶ JavaDoc para APIs públicas
- ▶ Inaceptables:
 - ▶ Los que mosquean (contradicen al código)
 - ▶ Código obsoleto comentado *por si acaso algun dia...* Usa git!
 - ▶ Comentarios de histórico de versiones
 - ▶ Los que no aportan nada

Comentarios en código

```
/** * Always returns true. */
```

```
public boolean isAvailable() {
```

```
    return false;
```

```
}
```

```
// somedev1 - 6/7/02 Adding temporary tracking of Login screen
```

```
// somedev2 - 5/22/07 Temporary my ass
```

```
return 1; // returns 1
```

```
// El día del mes
```

```
private int diaDelMes = 0
```

```
try {
```

```
.....
```

```
} catch (Exception e ) {
```

```
    // algo malo ha pasado
```

```
}
```



Formato

Formato Vertical

- ▶ Una clase (o fichero de procedures) no debería tener +200 líneas de media, nunca +500
- ▶ Metáfora del periodico:
 - ▶ Una clase empieza con un título descriptivo y sin detalles
 - ▶ Luego detalles de alto nivel
 - ▶ Mas abajo detalles de bajo nivel
 - ▶ Una clase son una mezcla de articulos mas y menos largos
 - ▶ Se deben entender “los titulares” sin tener que entrar al detalle

Formato

- ▶ Las variables se deben declarar cerca de su uso. Excepción: variables de clase al principio de la clase
- ▶ Anchura del tamaño de una pantalla, mejor no andar con scroll
- ▶ No romper sangrado aunque sólo haya una línea en un bloque
- ▶ Si una función invoca a otra, deben estar lo más cerca posible y la que llama estar por encima.
- ▶ Acordar entre el equipo cuales son las normas que se aplicarán a todo el código. **El equipo manda.**

Formato

- ▶ Evitar excesivo anidamiento. Si hasta hay que poner comentarios para saber cierres... es que hay un **grave** problema.

```
while ... {  
  if {  
    for {  
      if {  
[...]  
      }  
    } // for  
  }  
} // while
```



Objetos y estructuras de datos

Objetos y estructuras de datos

- ▶ Las clases, de cara al exterior, deben ocultar su estructura interna con abstracciones que operan sobre datos.
 - ▶ Las estructuras deben ocultar las operaciones y mostrar los datos
- ▶ El código orientado a procedimientos (que usa estructuras de datos) facilita la inclusión de nuevas funciones sin modificar las estructuras de datos existentes.
 - ▶ Código orientado a objetos facilita inclusión de nuevas clases sin cambiar funciones existentes

Objetos y estructuras de datos

- Las formas son estructuras de datos, sin comportamiento. Todo el comportamiento está en la clase Geometry
- Si añadimos un método *perimetro* las clases de formas (y sus descendientes) no se verían afectadas
- Si añadimos una nueva forma, hay que cambiar todos los metodos de la clase Geometry

Listado 6.5. Forma mediante procedimientos.

```
public class Square {
    public Point topLeft;
    public double side;
}

public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}

public class Circle {
    public Point center;
    public double radius;
}

public class Geometry {
    public final double PI = 3.141592653589793;

    public double area(Object shape) throws NoSuchShapeException
    {
        if (shape instanceof Square) {
            Square s = (Square)shape;
            return s.side * s.side;
        }
        else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle)shape;
            return r.height * r.width;
        }
        else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.radius;
        }
        throw new NoSuchShapeException();
    }
}
```


Objetos y estructuras de datos

- Metodo área polimórfico.
- No necesitamos una clase Geometry, cada clase sabe por si misma como comportarse.
- Si añadimos una nueva forma, no hay que tocar nada del código existente
- Pero por otro lado, si añadimos una nueva operación (perímetro), hay que tocar todas las formas existentes

Listado 6.6. Formas polimórficas.

```
public class Square implements Shape {
    private Point topLeft;
    private double side;

    public double area() {
        return side*side;
    }
}

public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;

    public double area() {
        return height * width;
    }
}

public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

    public double area() {
        return PI * radius * radius;
    }
}
```

Objetos y estructuras de datos

- ▶ Asi que tambien es cierto lo contrario:

- ▶ El código orientado a procedimientos dificulta la inclusión de nuevas estructuras de datos (es necesario cambiar todas las funciones)
- ▶ Código orientado a objetos dificulta inclusion de nuevas funciones porque es necesario cambiar todas las clases.

Objetos y estructuras de datos

- ▶ Ley de Demeter
 - ▶ Un módulo no debe conocer las interioridades de los objetos que manipula: estos deben ocultar su implementación a través de operaciones.
 - ▶ Un método ***m*** de una clase ***C*** solo debe invocar:
 - ▶ ***C***
 - ▶ Objetos creados por ***m***
 - ▶ Objetos pasado como argumentos a ***m***
 - ▶ Objetos variables de instancia de ***C***

Funciones - Argumentos de métodos

```
File cvsTempFile = new File(getConfig().getTempDir().getAbsolutePath() + "/" + System.currentTimeMillis() + ".cvs")  
File xlsTempFile = new File(getConfig().getTempDir().getAbsolutePath() + "/" + System.currentTimeMillis() + ".xls")
```

```
File tempFile = getConfig().getCvsTempFile()  
File tempFile = getConfig().getXlsTempFile()
```

```
File tempFile = getConfig().getTempFileWithExtension( "xls")
```

Objetos y estructuras de datos

- ▶ Data Transfer Objects (o, en realidad, beans)
 - ▶ Estructura de datos (get/sets). Con o sin representacion en BD
 - ▶ No deberían tener comportamiento.
- ▶ Tipo especial: Active Records
 - ▶ Mas parecido a clases de dominio Grails
 - ▶ Tienen save, find...
 - ▶ Podemos caer en la tentación de añadirles metodos con reglas de negocion. ¡Error!



Procesamiento de errores

Errores

- ▶ Usar excepciones en lugar de códigos de error

```
if( condicionError) {  
    return -1  
} else if (otraCondicion ) {  
    return -2  
} else {  
    return 0  
}
```

- ▶ Esto obliga a quien lo llama a verificar inmediatamente la condición

Errores

- ▶ (Cosecha propia)
 - ▶ Manejar solo las excepciones que sepas manejar. ¿Como vas a manejar un error de conectividad a BD. ¡Lanzalo!
 - ▶ No hacer *gili-catches*

```
try{  
} catch {  
    // nada por aqui  
}
```


Errores

- ▶ No todas las exceptions tienen que ser *checked* . Si lo hacemos así, obligamos a que todo aquel que llame cambie su firma de clase para verificar.
- ▶ A veces puede ser necesario pero por ejemplo en Groovy la mayoría son *Unchecked* (o `RuntimeException`), así solo las *catcheas* cuando lo necesites
- ▶ Incluir el contexto donde se produjo la exception: que el mensaje de error sea informativo (el `stackTrace` ya lo tenemos)

Errores

- ▶ Si una libreria de terceros nos devuelve muchas exceptions puede ser conveniente hacer un envoltorio
- ▶ No devolver **null**. Obliga a hacer comprobaciones constantemente y puede provocar NPE facilmente (¡en ejecución!)
- ▶ No pasar **null**. Seguro que tenemos un if lo primero. Si queremos verificar parámetros, tal vez sea bueno una verificación que ante cualquier problema salte un `InvalidArgumentException`, o asserts



Limites

Limites

- ▶ A veces las clases del sistema o généricas son demasiado “grandes” para lo que necesitamos
- ▶ Usarlas lo menos posible como valores de retorno en nuestros API (¿y si cambian y nos estropean algo?)

```
Map libros = new HashMap()  
Libro libro = new Libro(...)  
libros.put( id , libro)  
...  
Libro libro2 = (Libro)libros.get(id)
```

```
Map<Libro> libro = new HashMap<Libro>()
```

Limites

```
Class MapaDeLibros {  
    private Map libros = new HashMap<Libro>()  
  
    public Libro addLibro( String clave, Libro libro ){...}  
  
    public Libro getLibro( String clave ){  
        return libros.get(clave)  
    }  
    // o, incluso, estilo antiguo pero al menos en un solo sitio  
    public Libro getLibro( String clave ){  
        return (Libro)libros.get(clave)  
    }  
}
```



Pruebas unitarias

TDD

- ▶ Hasta el año 1997 no existía el concepto de TDD.
- ▶ Las pruebas eran mini-programas que luego tirábamos a la basura
- ▶ **Leyes de TDD**
 - ▶ No se debe crear código hasta que no haya un test que falle
 - ▶ El código de prueba tiene que ser el mínimo para que el código de producción falle
 - ▶ El código de producción tiene que ser el mínimo para que el test no falle
- ▶ Estas reglas garantizan que el código y los test se crean en paralelo

TDD

- ▶ Los test son indispensables en el desarrollo moderno de software.
- ▶ Garantizan que se pueden hacer cambios en el futuro sin romper nada
- ▶ Son ejemplos de uso del código. No ignorar los test aparentemente triviales porque son útiles como ejemplos
- ▶ En los test, es más importante aún la legibilidad. No desaproveches la ocasión de crear muchas variables si aumenta la legibilidad o penaliza el rendimiento
- ▶ Evitar métodos muy largos con muchos detalles de implementación

TDD

- ▶ Los test deben ser F.I.R.S.T
 - ▶ **F**ast
 - ▶ **I**ndependent
 - ▶ **R**epetition
 - ▶ **S**elf-Validating
 - ▶ **T**imely
- ▶ No tener pruebas es muy malo. Pero tener malas pruebas es peor
- ▶ El código de test no se tira: es tan importante como el “de verdad”. Nos quitan el miedo a hacer pruebas (junto con control de versiones)



Classes

Clases

- ▶ Orden dentro de la clase:
 - ▶ Constantes estáticas
 - ▶ Variables estáticas
 - ▶ Variables de instancia
 - ▶ Métodos/Funciones.

De todo ello, primero lo público y después lo privado.

- ▶ Tamaño reducido
- ▶ Tamaño aún más reducido

S.O.L.I.D

▶ Single Responsibility

- ▶ Una sola (y simple) responsabilidad por clase. Solo un motivo para cambiar la clase.
- ▶ Evitar el “ya que estoy aqui, meto esto tambien”
- ▶ Si conceptualmente lo que va a hacer es otra cosa, sácalo a otra clase

▶ Open / Closed

- ▶ Una clase debe estar abierta a extension pero cerrado a modificaciones
- ▶ La extensión mas habitual es herencia, pero tambien la composición puede ser util.

▶ Liskov Substitution

- ▶ Una clase hija se debe poder usar en lugar de una padre: no reimplementar métodos que rompan funcionamiento superior

▶ Interface Segregation

- ▶ Los interfaces deben tener un sentido concreto y finito: mejor muchos interfaces pequeños a pocos grandes

▶ Dependency Inversion



Sistemas

- ▶ Para dedicar una charla entera (o leerse el capítulo 11 del libro):
 - ▶ Factorias abstractas
 - ▶ Inyeccion de dependencias
 - ▶ AOP
 - ▶ Proxies
 - ▶ DSL



Diseño sencillo

Diseño sencillo

- ▶ Reglas de Kent Beck para diseño sencillo

- ▶ Que pase todas las pruebas
- ▶ No hay código duplicado
- ▶ Expresa la intención del programador
- ▶ Minimiza el número de clases y métodos

← Refactorizar

Hacemos un cambio,
pasa los tests, paramos
un segundo y pensamos:
¿es mejor que antes?

Eliminar duplicados

- ▶ DRY!
- ▶ No solo por comodidad: nos ayudan a reducir la complejidad
- ▶ Ayudan a pensar como mejorar el código
- ▶ Si tienes buenos test no tienes que tener miedo a romper nada

Expresividad

- ▶ Cualquier idiota puede hacer código que compila
- ▶ Solo un buen programador puede hacer código que otros entiendan
- ▶ El mayor coste del software es el mantenimiento a largo plazo
- ▶ Elegir nombres adecuados para metodos, funciones y variables.



Concurrencia

Concurrencia

- ▶ Capítulo (13) tambien para dedicarle un tiempo aparte.
- ▶ **Notitas:**
 - ▶ Multiproceso no siempre mejora rendimiento: solo si tenemos varios procesadores o máquinas
 - ▶ El diseño del programa cambia si tenemos concurrencia
 - ▶ Muchas veces es mas que conveniente usar objetos inmutables.
 - ▶ Es necesario conocer la infraestructura subyacente (contenedor web, ejbs..)
 - ▶ Saber que clases admiten sincronizacion y cuales no (Hashtable, Hashmap...)
 - ▶ Puede ser conveniente separar el código concurrente del resto. Y que sea lo más pequeño posible.
 - ▶ Usar colas: Product-Consumer o Publish-Subscribe para desacoplar
 - ▶ Bloqueos: controlar, provocar y prevenir
 - ▶ Si usamos *synchronized* que sean bloques lo mas pequeños posibles para prevenir deadlocks

Suficiente por hoy!

Esto ha sido una introducción.

Recomendación: leeros el libro!. Ved mas videos y slides

Preguntas?

Suficiente por hoy!

Esto ha sido una introducción.

Recomendación: leeros el libro!. Ved mas videos y slides

Preguntas?

Suficiente por hoy!

Esto ha sido una introducción.

Recomendación: leeros el libro!. Ved mas videos y slides

Preguntas?

Suficiente por hoy!

Esto ha sido una introducción.

Recomendación: leeros el libro!. Ved mas videos y slides

Preguntas?

Let's start with the first set of slides

The background features a dark, moody photograph of a laptop and a notebook. A black pen with the word 'PRECISE' and a logo is resting on the notebook. A large, bright orange geometric shape, resembling a stylized 'A' or a series of overlapping triangles, is positioned on the right side of the image. The text 'THIS IS YOUR PRESENTATION TITLE' is written in a clean, white, sans-serif font, centered on the left side of the image.

THIS IS YOUR
PRESENTATION
TITLE

Instructions for use

Open this document in Google Slides (if you are at [slidescarnival.com](https://www.slidescarnival.com) use the button below this presentation)
You have to be signed in to your Google account

EDIT IN GOOGLE SLIDES

Go to the **File** menu and select **Make a copy**. You will get a copy of this document on your Google Drive and will be able to edit, add or delete slides.

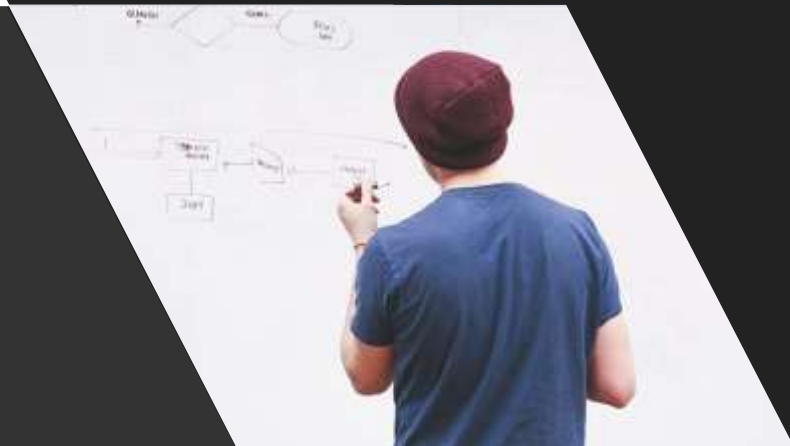
EDIT IN POWERPOINT®

Go to the **File** menu and select **Download as Microsoft PowerPoint**. You will get a .pptx file that you can edit in PowerPoint.

Remember to download and install the fonts used in this presentation (you'll find the links to the font files needed in the [Presentation design slide](#))

More info on how to use this template at www.slidescarnival.com/help-use-presentation-template

This template is free to use under [Creative Commons Attribution license](#). You can keep the Credits slide or mention SlidesCarnival and other resources used in a slide footer.



HELLO!

I am Jayden Smith

I am here because I love to
give presentations.

You can find me at
@username

1.

Transition headline

Let's start with the first set of slides

“

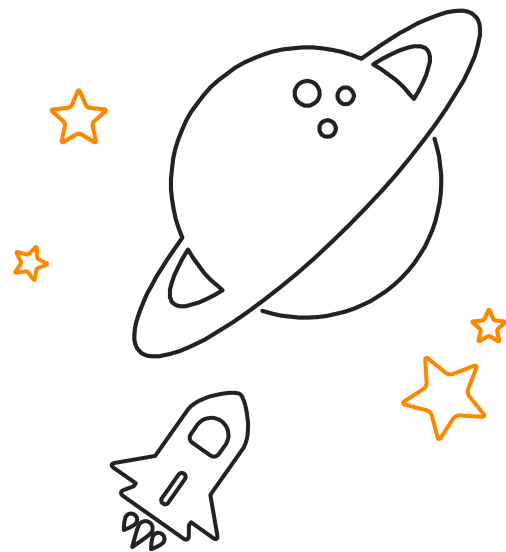
Quotations are commonly printed as a means of inspiration and to invoke philosophical thoughts from the reader.

”

This is a slide title

- ▶ Here you have a list of items
- ▶ And some text
- ▶ But remember not to overload your slides with content

You audience will listen to you or read the content, but won't do both.



BIG CONCEPT

Bring the attention of your audience over a key concept using icons or illustrations

You can also split your content

White

Is the color of milk and fresh snow, the color produced by the combination of all the colors of the visible spectrum.

Black

Is the color of coal, ebony, and of outer space. It is the darkest color, the result of the absence of or complete absorption of light.

In two or three columns

Yellow

Is the color of gold, butter and ripe lemons. In the spectrum of visible light, yellow is found between green and orange.

Blue

Is the colour of the clear sky and the deep sea. It is located between violet and green on the optical spectrum.

Red

Is the color of blood, and because of this it has historically been associated with sacrifice, danger and courage.

A picture is worth a thousand words

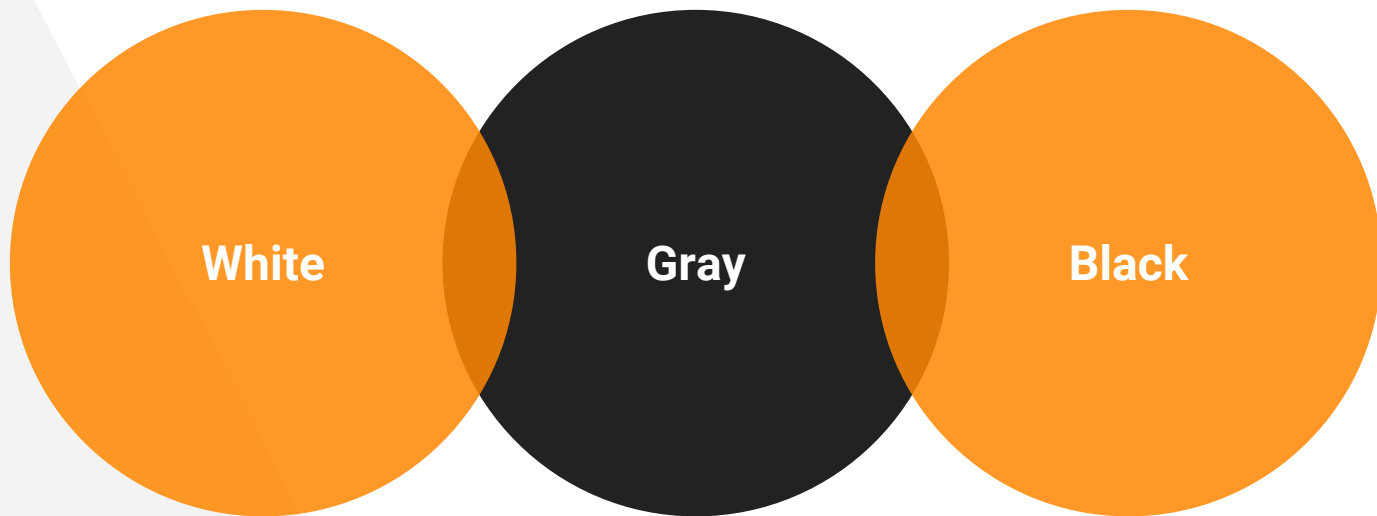
A complex idea can be conveyed with just a single still image, namely making it possible to absorb large amounts of data quickly.



Want big impact? USE BIG IMAGE

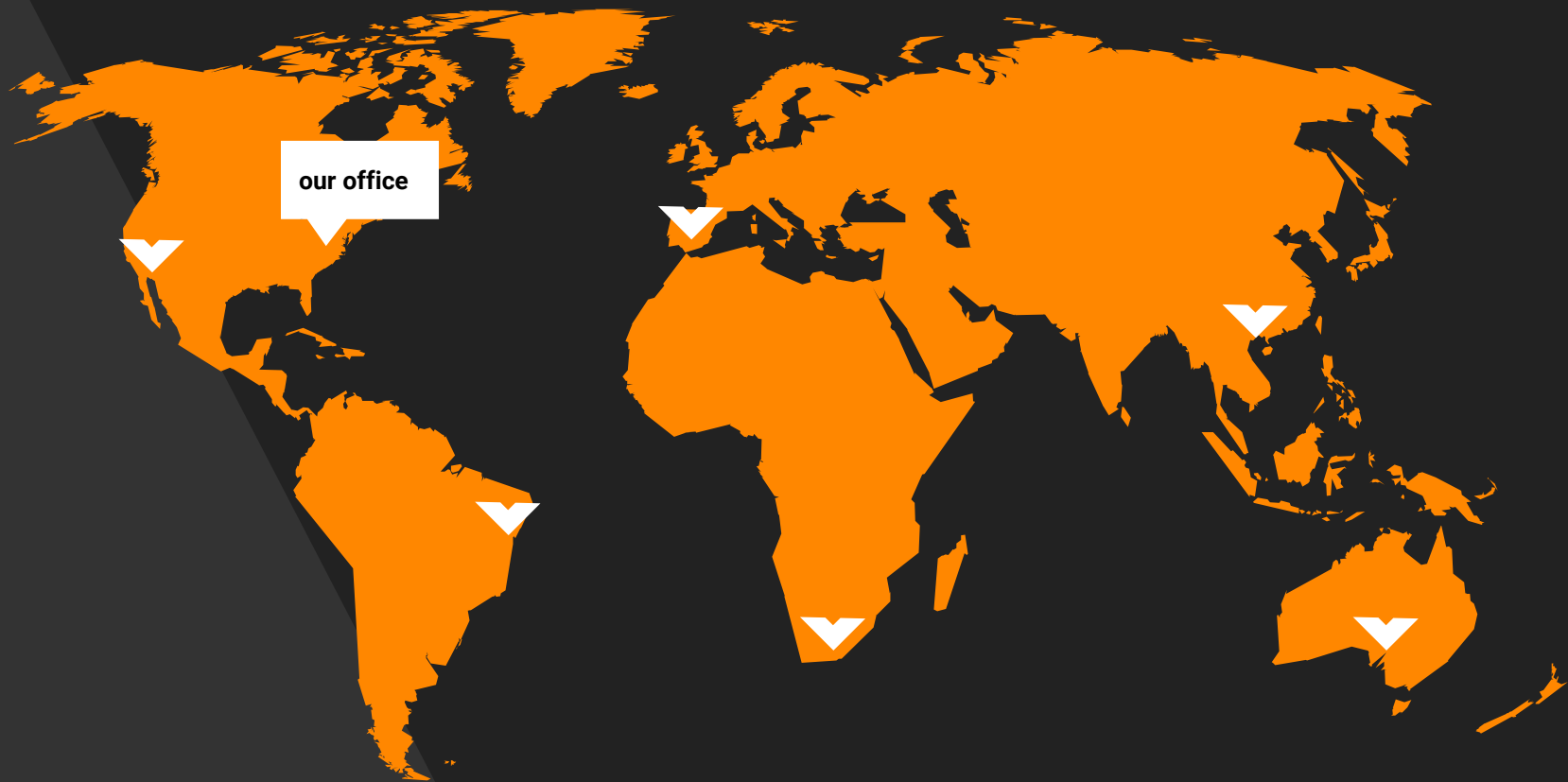


Use charts to explain your ideas



And tables to compare data

	A	B	C
Yellow	10	20	7
Blue	30	15	10
Orange	5	24	16



89,526,124

Whoa! That's a big number,
aren't you proud?

89,526,124\$

That's a lot of money

185,244 users

And a lot of users

100%

Total success!

Our process is easy



Let's review some concepts

Yellow

Is the color of gold, butter and ripe lemons. In the spectrum of visible light, yellow is found between green and orange.

Blue

Is the colour of the clear sky and the deep sea. It is located between violet and green on the optical spectrum.

Red

Is the color of blood, and because of this it has historically been associated with sacrifice, danger and courage.

Yellow

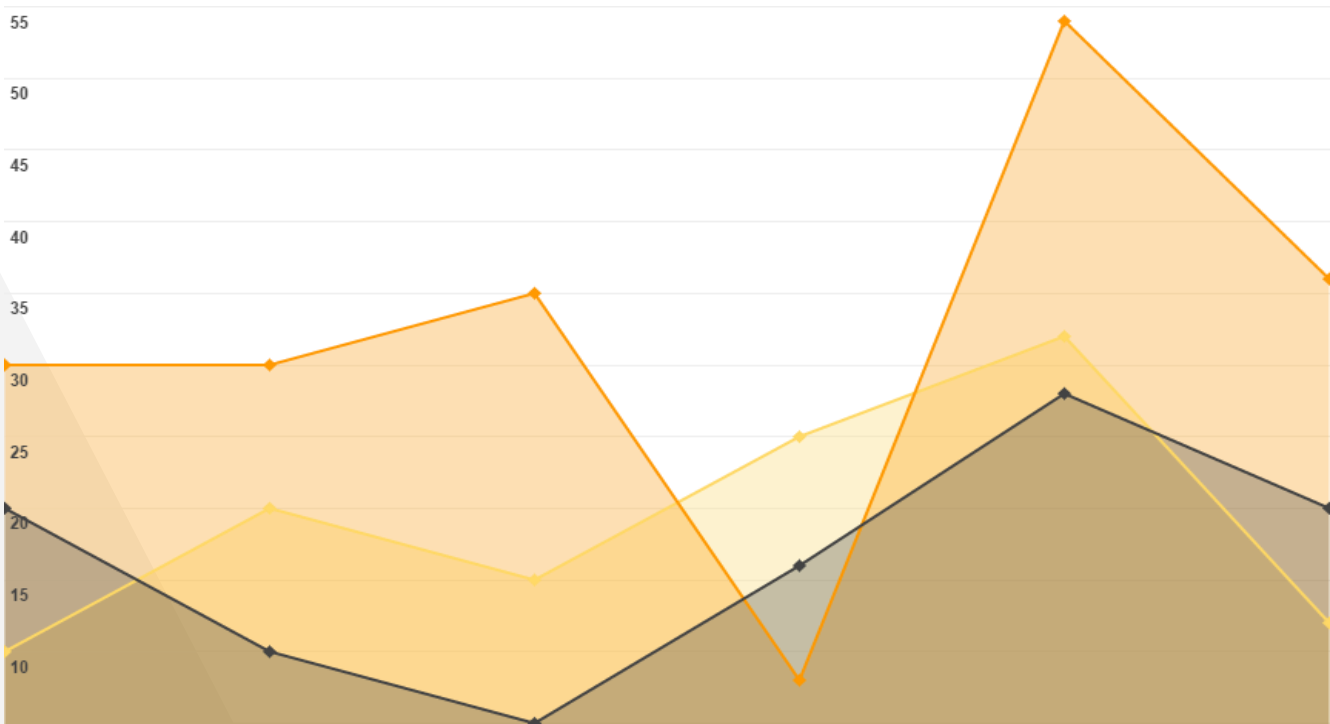
Is the color of gold, butter and ripe lemons. In the spectrum of visible light, yellow is found between green and orange.

Blue

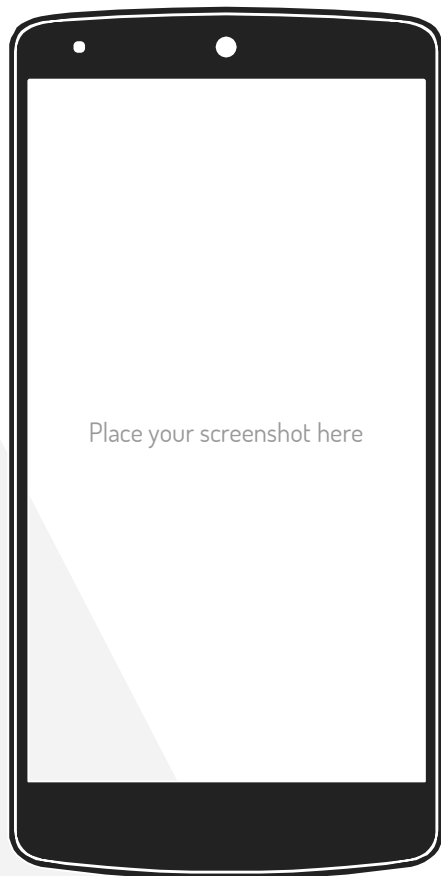
Is the colour of the clear sky and the deep sea. It is located between violet and green on the optical spectrum.

Red

Is the color of blood, and because of this it has historically been associated with sacrifice, danger and courage.

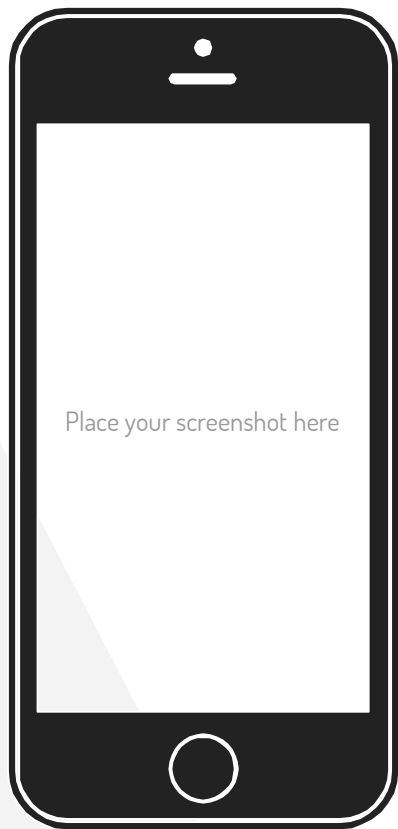


You can insert graphs from [Google Sheets](#)



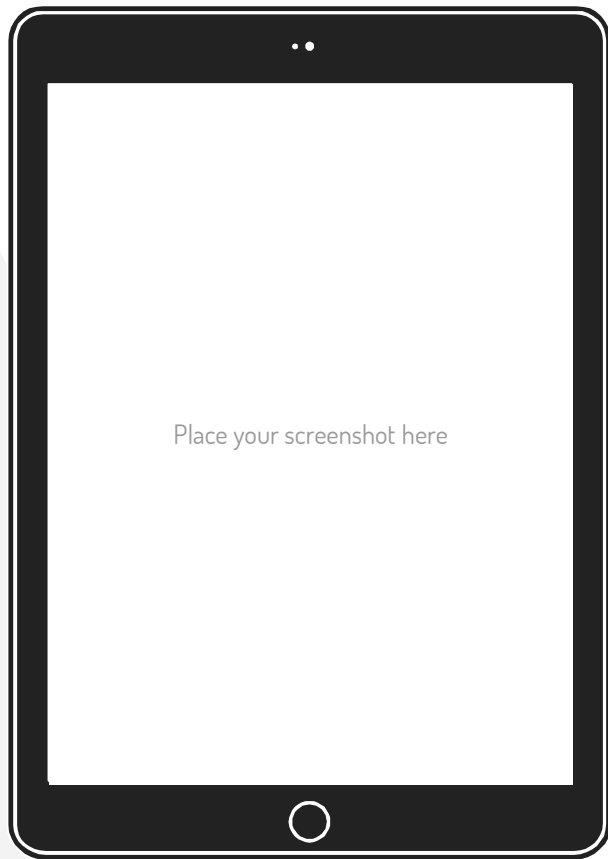
ANDROID PROJECT

Show and explain your web, app or software projects using these gadget templates.



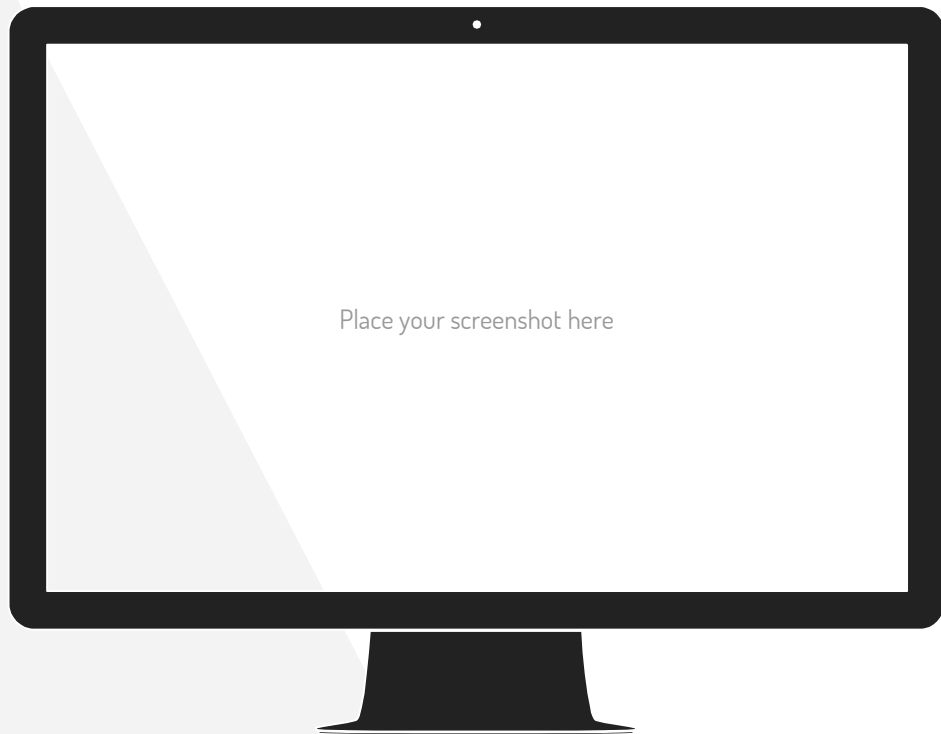
iPHONE PROJECT

Show and explain your web, app or software projects using these gadget templates.



TABLET PROJECT

Show and explain your web, app or software projects using these gadget templates.



DESKTOP PROJECT

Show and explain your web, app or software projects using these gadget templates.

THANKS!

Any questions?

You can find me at @username & user@mail.me

Credits

Special thanks to all the people who made and released these awesome resources for free:

- ▶ Presentation template by [SlidesCarnival](#)
- ▶ Photographs by [Startupstockphotos](#)

Presentation design

This presentation uses the following typographies and colors:

- ▶ Titles: **Dosis**
- ▶ Body copy: **Roboto**

You can download the fonts on these pages:

<https://www.fontsquirrel.com/fonts/dosis>

<https://material.google.com/resources/roboto-noto-fonts.html>

- ▶ Orange **#ff8700**

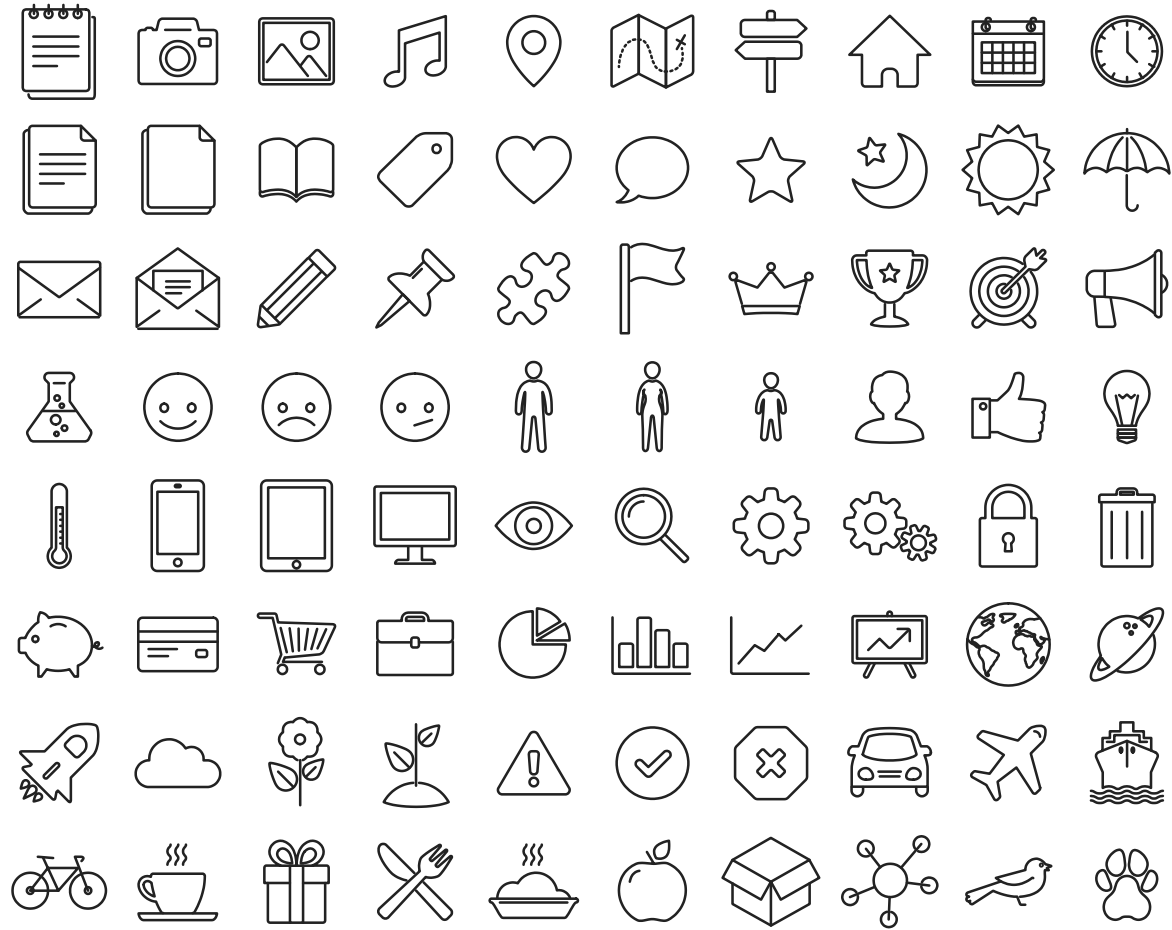
You don't need to keep this slide in your presentation. It's only here to serve you as a design guide if you need to create new slides or download the fonts to edit the presentation in PowerPoint®

SlidesCarnival icons are editable shapes.

- This means that you can:
- Resize them without losing quality.
 - Change line color, width and style.

Isn't that nice? :)

Examples:





Now you can use any emoji as an icon!

And of course it resizes without losing quality and you can change the color.

How? Follow Google instructions

<https://twitter.com/googledocs/status/730087240156643328>

