User code context

"Context" here means some place in framework which calls user code. For example if you use Template Method pattern, context will be a method in the parent class which calls template method. The term "context" in this meaning is just something I came up with and it is not used in Wicket javadocs. As far as javadocs concerned *event handling context* corresponds to *event handling phase* and *rendering context* corresponds to *response phase*.

There are three common contexts in Wicket from which user code may be called (numeration here implies the order in which code will be called):

1. page creation
2. event handling (optional)
3. rendering

```
public class MyPage extends WebPage ...
    public MyPage() {
        ...
        // 1 - page creation
        final Form form = new Form("form", model) {
            @Override
            protected void onSubmit() {
                // 2 - event handling (optional)
            }

            @Override
             protected void onBeforeRender() {
                // 3 - rendering
                super.onBeforeRender();
             }
        };
        add(form);
    }
```

Page creation

Page creation means executing code in page constructor. Page creation occurs in two cases:

- page is created by Wicket using IPageFactory as a part of request processing. (How often page instance of certain class is created depends on page type. See Pages for more details.)
- page is created by user code using `new` keyword

There are no restrictions on what you can do in page creation context. The common thing is to create components and models.

Beware of calling overriden methods in constructor, since at the time overriden method is called in base class constructor, child class which overrides this method, is not fully constructed. If you call overriden methods in constructor anyway, make sure they do not use properties from child class.

Event handling

Event handling is a process of executing code in methods like onSubmit(), onClick() and the like. All these methods calls originates from implementations of IRequestListener interface (see IRequestListener subinterfaces for the full list of all possible callbacks). Event handling may happen as many times as event occurs (for example user produces some action). Normally event handling happens in another request than the one which created the page.
There are no restrictions on what can be done in event handling context. Although the "usual" thing is to perform some king of business logic, you can also create/change/add/remove components. Since event handling precedes rendering, any changes to components and model will be "visible" in rendering context.

Replacing panel in event handler

```
public class MyPage extends WebPage ...
    public MyPage() {
        ...
        // note that panels use the same id
        add(new Link("feedback") {
            public void onClick() {
                MyPage.this.replace(new FeedbackPanel("mainPanel"));
            }
        });
        add(new Link("about") {
            public void onClick() {
                MyPage.this.replace(new AboutPanel("mainPanel"));
            }
        });
    }
```

Rendering

Rendering is performed after page creation and event handling. It happens as many times as page or components (in case of ajax) is requested by browser. Whether rendering is performed in the same request as event handling depends on rendering strategy.
Rendering consists of three parts which have different contexts:

- before render (onBeforeRender() method). It's called on all visible components, unless they override callOnBeforeRenderIfNotVisible(). There are no restrictions on what can be done in this context.

  Using page's onBeforeRender() for delayed configuration

  ```
  public class MyPage extends WebPage ...
      private final Label smartLabel;
      private final Label label;
      private boolean useSmartLink;

      public MyPage() {
          ...
          // note that labels use the same id
          smartLabel = new SmartLinkLabel("label");
          label = new Label("label");
      }

      public void setUseSmartLink(final boolean useSmartLink) {
          this.useSmartLink = useSmartLink;
      }

      protected void onBeforeRender() {
          if (useSmartLink)
              addOrReplace(smartLink);
          else
              addOrReplace(link);

          super.onBeforeRender();
      }
  ```

- component rendering (IModel#getObject(), onComponentTag(), onComponentTagBody() methods). To distinct between rendering on the whole and the part of rendering which produces markup, *"component rendering"* is used here to refer to markup creating.
  Component rendering is performed only for visible components and is not allowed to change components state or models.
  Reason for not changing components and models during rendering is that they maybe shared between components and changing them may lead to inconsistent results. In the example below two labels use the same model which is changed during rendering of the second label. The consequence is that output of the first label depends entirely on the order in which labels appear in markup file.

Changing model during component rendering. Result depends on markup. Don't do that.

```
public class MyPage extends WebPage ...
     public MyPage() {
        ...

        final IModel model = new Model("---first label text---");

        final Label label = new Label("label", model);
        final Label label2 = new Label("label2", model) {
            protected void onComponentTag(final ComponentTag tag) {
                super.onComponentTag(tag);
                model.setObject("---second label text---");
            }
        };
     }
```

In most cases component will throw exception if it detects that it's being changed. Though no exception will be thrown in case of calling is /setEnabled() and some other methods when they are called on not versioned component (see Component versioning). There are also no warnings if model is changed using IModel#setObject() method.

- after rendering (onAfterRender() method). It's called on all components regardless their visibility. As component rendering it should not change components state or models.

Models

Model's methods are called by Wicket in some of the above contexts. Generally, you don't need to think when model methods are called and what rules apply unless you are writing some clever code inside a model. This example code shows in comments when model methods may be called:

```
public class MyPage extends WebPage ...
    public MyPage() {
        ...
        final Model model = new Model() {
            public Object getObject() {
                // called on event handling and component rendering
                return super.getObject();
            }

            public void setObject(final Serializable object) {
                // called on event handling
                super.setObject(object);
            }
        };
        final TextField text = new TextField("text", model);
    }
```