

Shell Commands

man ARG *show manual page*. Zeigt die man page für Programm **ARG** an.

cd ARG *change directory*. Wechselt ins Verzeichnis **ARG**.

pwd *print working directory*. Zeigt das aktuelle Verzeichnis an.

ls *list directory*. Zeigt den Inhalt des aktuellen Verzeichnisses an.

mv ARG1 ARG2 *move*. Verschiebt die Datei **ARG1** zu **ARG2**.

cp ARG1 ARG2 *copy*. Kopiert die Datei **ARG1** zu **ARG2**.

rm ARG *remove*. Löscht die Datei **ARG**.

touch ARG Erstellt eine leere Datei namens **ARG**.

mkdir ARG *make directory*. Erstellt ein Verzeichnis namens **ARG**.

cat ARG Zeigt den Inhalt von Datei **ARG** an.

ps *list processes*. Zeigt die aktuell laufenden Prozesse an.

find Suche nach Dateien.

grep *globally search a regular expression and print*. Nützlich um Sachen zu durchsuchen.

Redirecting Output

| (**pipe**) Leitet den Output von einem Programm in eine anderes Programm.

> Leitet den Output von einem Programm in eine Datei.

< Leitet den Inhalt einer Datei in ein Programm.

Listing 1: Zeige alle Prozesse an (**ps -ef**) und filtere danach nach bash (**grep bash**).

\$ ps -ef grep bash
david 5333 5146 0 10:15 pts/4 00:00:00 bash
david 7069 1 0 14:46 ? 00:00:00 /bin/bash
david 8306 1 0 15:00 ? 00:00:00 /bin/bash

Listing 2: Zeige alle Prozesse an (**ps -ef**) und speichere die Ausgabe in der Datei **tmp**.

\$ ps -ef > tmp

Git Ablauf

Einmalig: Git konfigurieren (Benutzername, Email).

Einmal pro repository und PC: Klonen (**clone**).

Normaler Arbeitsablauf:

- Bevor man etwas ändert: **pull** der aktuellen Version vom Server.
- Am Projekt arbeiten.
- Wenn man mit etwas fertig ist:
 - Anschauen welche Dateien geändert wurden (**status**).
 - Commit vorbereiten, indem man alle Dateien die zum commit gehören hinzufügt (**add**).
 - **commit** ausführen, und eine Nachricht angeben was man gemacht hat.
 - Bemerkung: Commits sind nur lokal, solange man nicht **push** ausführt.
- Wenn man etwas auf den Server laden will: **push**. Bemerkung: **push** lädt alle commits auf den Server. Wurde seit dem letzten **pull** etwas geändert, muss zuerst noch ein **pull** durchgeführt werden.

Bemerkung: Diese Befehle (normaler Arbeitsablauf) müssen nicht in dieser Reihenfolge ausgeführt werden, z.b. kann auch mehrmals **add** aufgerufen werden bevor man einen **commit** macht. Oder man kann mehrere **commits** auf einmal auf den Server **pushen**.

Git Commands

Hilfe zu den einzelnen commands kann man über die **man** page erhalten. Beachte dass bei git die man pages direkt nach den commands aufgeteilt ist, d.h. will man Hilfe für den Befehl **git add** erhalten muss man **man git-add** eingeben (git und der jeweilige Befehl sind immer durch ein - getrennt).

git clone URL Klont das *repository* vom server von **URL**.

git add FILE1 FILE2 ... Fügt die Dateien **FILE1, FILE2, ...** zum nächsten *commit* hinzu.

git status Zeigt die veränderten Dateien seit dem letzten *commit* an.

git commit Macht einen *commit* mit allen Dateien die hinzugefügt wurden.

git push Pusht alle lokalen *commits* auf den Server. (\approx "upload")

git pull Lädt die neusten *commits* vom Server sind runter.

Git Konzept

Bei git gibt es immer *branches*. Im einfachsten Szenario (meistens völlig ausreichend) hat man einen lokalen branch (die Kopie vom Server), und einen remote branch (der Server). Diese branches werden üblicherweise **master** genannt, z.b. ist **master** der Name des lokalen branch, und **origin/master** der branch auf dem Server.

push und **pull** sorgen dafür dass, diese beiden branches abgeglichen werden. Ein branch ist einfach eine Abfolge von commits, also einfach eine Serie von Änderungen welche gruppiert wurden.

Aufgabe 1

Mit git arbeitet man immer mit einem *repository*. Einfach gesagt ist das repository der “Server-Teil” von einem git Projekt. Wir erstellen wir ein repository auf GitHub, da wir diesen Service verwenden.

Bemerkung: Git \neq GitHub. GitHub ist nicht git, GitHub ist nur ein Dienstleister, welcher gratis *git repositories* anbietet (gratis, solange die repositories öffentlich sind).

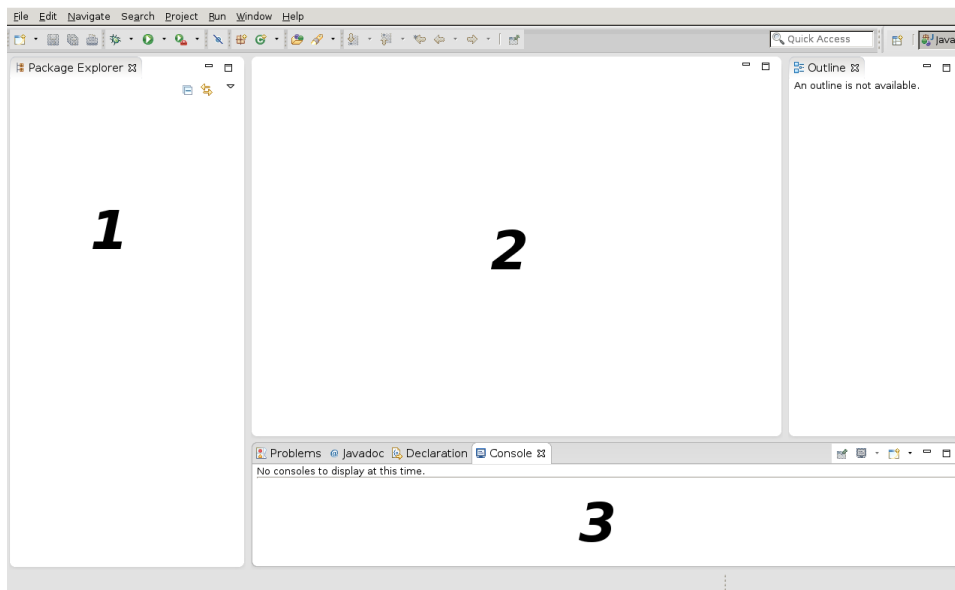
Auf GitHub ist beschrieben, wie genau das erste mal git konfiguriert werden soll und wie das repository erstellt wird etc.: Siehe <https://help.github.com/articles/set-up-git>

Aufgabe: Die Anleitung <https://help.github.com/articles/set-up-git> durcharbeiten.

Aufgabe 2

Im erstellten Repository ein Projekt mit Eclipse anlegen. Einfaches Logging implementieren, anhand des vorgegebenen Templates (Link wird noch bekannt gegeben).

Eclipse Übersicht



- 1 Hier werden die Projekte angezeigt und die jeweiligen Dateien (z.B. Klassen) der Projekte.
- 2 Hier kann man Dateien editieren.
- 3 Die Konsole. Hier wird der Input/Output des Programms angezeigt sobald es ausgeführt wird.

Ich sehe eine der Ansichten nicht, wie kann ich sie einblenden?

Im Menu unter **Window** > **Show View** > ... können alle verschiedenen Fenster eingeblendet werden, falls sie geschlossen wurden.

Projekte

In Java arbeitet man immer mit *Projekten*. Ein Projekt entspricht einem Programm, d.h. solange Sie an *einem* Programm arbeiten, arbeiten Sie *immer im gleichen Projekt*.


Wenn Sie mit anderen Leuten zusammen arbeiten, so muss das Projekt nur *einmal* erstellt werden. Der Arbeitsablauf um ein Projekt zu erschaffen, und dass dann alle Gruppenmitglieder das Projekt in ihrem Eclipse haben ist wie folgt (**Achtung: Die Schritte müssen in dieser Reihenfolge ausgeführt werden!**):

1. Jemand erstellt das Projekt bei sich in Eclipse. Nur diese Person *erstellt* ein Projekt! (Siehe "Ein neues Projekt anlegen (im Git Repository)")
2. Die gleiche Person lädt das Projekt auf den Server. (Siehe "Das erstellte Projekt pushen")
3. Jetzt sind die anderen Gruppenmitglieder dran: Sie updaten ihr repository (**git pull**) und importieren das Projekt in Eclipse. (Siehe "Das Projekt importieren")

Ein neues Projekt anlegen (im Git Repository)

Wenn Sie ein neues Projekt anlegen wollen können sie das über das Menu **File > New > Java Project** machen.

Create a Java Project
Enter a project name.



Project name: **1**

☐ Use default location

Location: **2**

JRE

☒ Use an execution environment JRE:

☐ Use a project specific JRE:

☐ Use default JRE (currently 'java-7-openjdk-amd64') [Configure JREs...](#)

Project layout

☐ Use project folder as root for sources and class files

☒ Create separate folders for sources and class files [Configure default...](#)

Working sets

☐ Add project to working sets

Working sets:

- 1** Geben Sie dem Projekt einen Namen. Wenn Sie z.B. das Spiel "Space Invaders" programmieren, würden Sie das Projekt "SpaceInvaders" nennen (wenn Sie auf Leerzeichen (white-spaces) verzichten macht das Ihr Leben deutlich einfacher!).
- 2** Weil wir mit Git arbeiten erstellen wir das Projekt *nicht* in der **default location**, sondern wählen das Verzeichnis auf der Festplatte aus, wo wir unser Git Repository erstellt haben.

Ich möchte mehrere Projekte im gleichen Git Repository haben

Wenn Sie mehrere Projekte im gleichen Git Repository haben wollen, dann müssen Sie einen Unterordner im Repository erstellen und dann das Projekt darin anlegen.

Das erstellte Projekt pushen

Sie haben nun das Projekt auf Ihrem Dateisystem im Git Repository erstellt – doch jetzt wollen Sie es auch noch uploaden, damit Ihr Projekt Partner das erstellte Projekt auch bei sich hat.

```
--|bash|~|-----
|_|david|_|x201|-$ cd lee/lee/

--|bash|~|lee/lee|-----
|_|david|_|x201|-$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .classpath
        .project

nothing added to commit but untracked files present (use "git add" to track)

--|bash|~|lee/lee|-----
|_|david|_|x201|-$ git add .classpath .project

--|bash|~|lee/lee|-----
|_|david|_|x201|-$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .classpath
        new file:   .project

--|bash|~|lee/lee|-----
|_|david|_|x201|-$ git commit -m "Projekt angelegt."
[master aecab9b] Projekt angelegt.
2 files changed, 23 insertions(+)
create mode 100644 .classpath
create mode 100644 .project

--|bash|~|lee/lee|-----
|_|david|_|x201|-$ git push
Username for 'https://github.com': stolzda
Password for 'https://stolzda@github.com':
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 654 bytes | 0 bytes/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To https://github.com/stolzda/lee.git
246a6ec..aecab9b master -> master
```

cd REPOSITORY LOCATION Sie gehen ins Verzeichnis in dem sie das Repository geklont haben, hier wurde es nach “lee/lee” geklont.

git status Sie sehen, welche Dateien sich seit dem letzten **commit** geändert haben. Da wir das Projekt neu angelegt haben, gibt es zwei neue Dateien: **.classpath** und **.project**. Diese müssen beide committed werden damit jemand anders das Projekt direkt bei Eclipse importieren kann.

git add .classpath .project Wir fügen die beiden Dateien zum nächsten commit hinzu.

git status Wir schauen wir der Zustand unserer Kopie des Repositories jetzt ist. Wir sehen dass die beiden Dateien für den nächsten commit vorgemerkt sind.

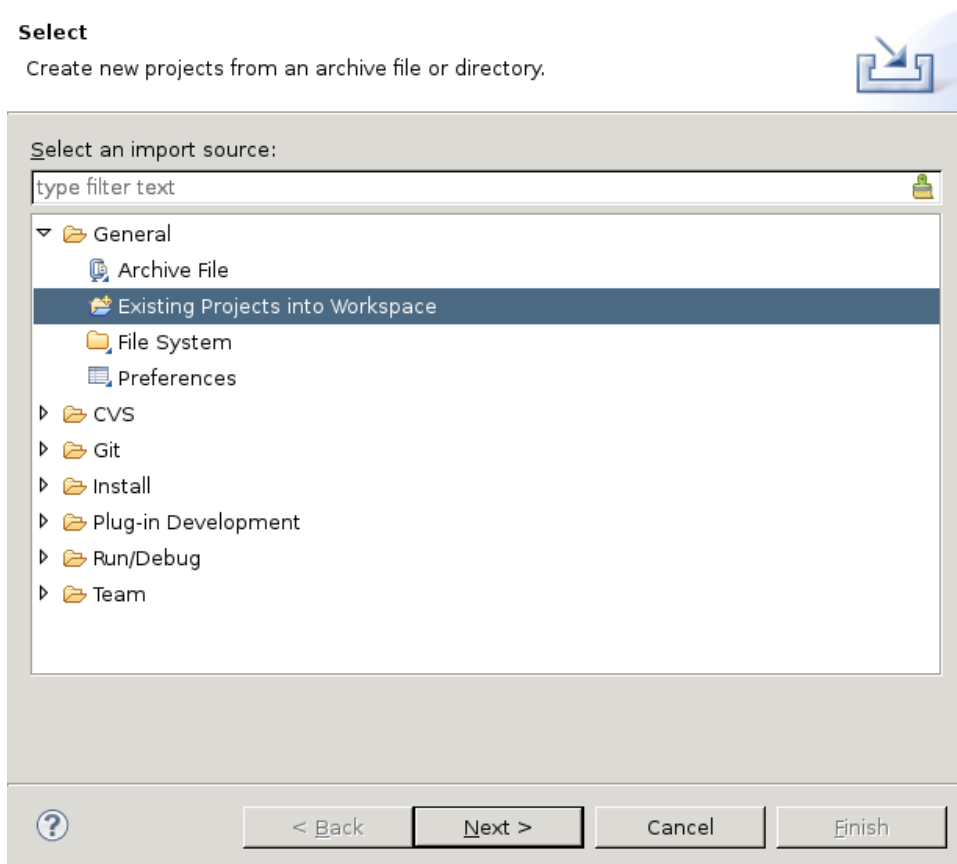
git commit -m “Projekt angelegt.” Wir erstellen einen commit mit der Nachricht “Projekt angelegt.”, damit wir später noch wissen was wir in diesem commit gemacht haben.

git push Da der commit bis jetzt nur lokal war, laden wir den commit auch auf den Server. Dazu pushen wir den commit.

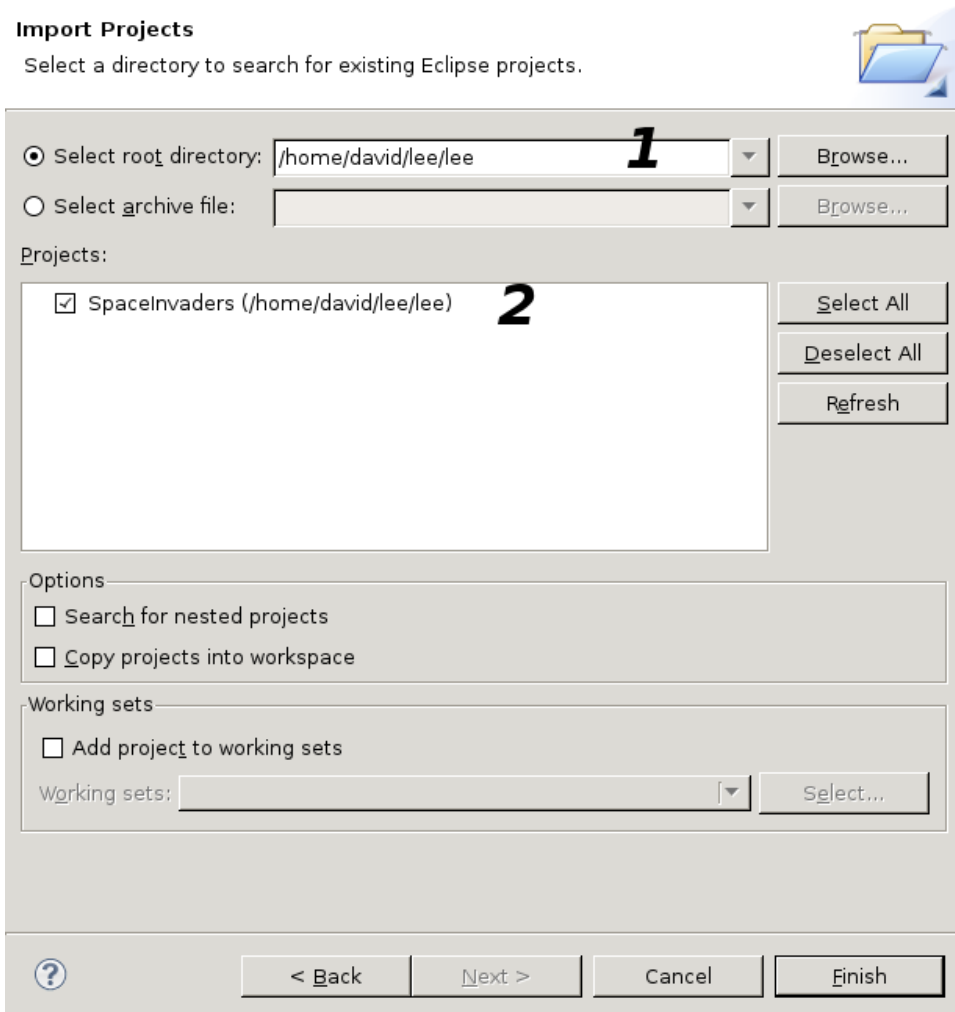
Das Projekt importieren

Wenn Ihr Projektpartner das Projekt in Eclipse angelegt hat und auf das Repository gepushed hat, können Sie das Projekt runterladen, in dem Sie normal ein **git pull** ausführen. Danach können Sie das Projekt in Eclipse *importieren*. Das kann über **File > Import...** gemacht werden.

Achtung: Erstellen Sie nicht ein neues Projekt, sonst haben Sie zwei verschiedene Projekte, anstatt dass Sie am gleichen Projekt arbeiten!



Im Wizard wählen Sie **Existing Projects into Workspace** aus.

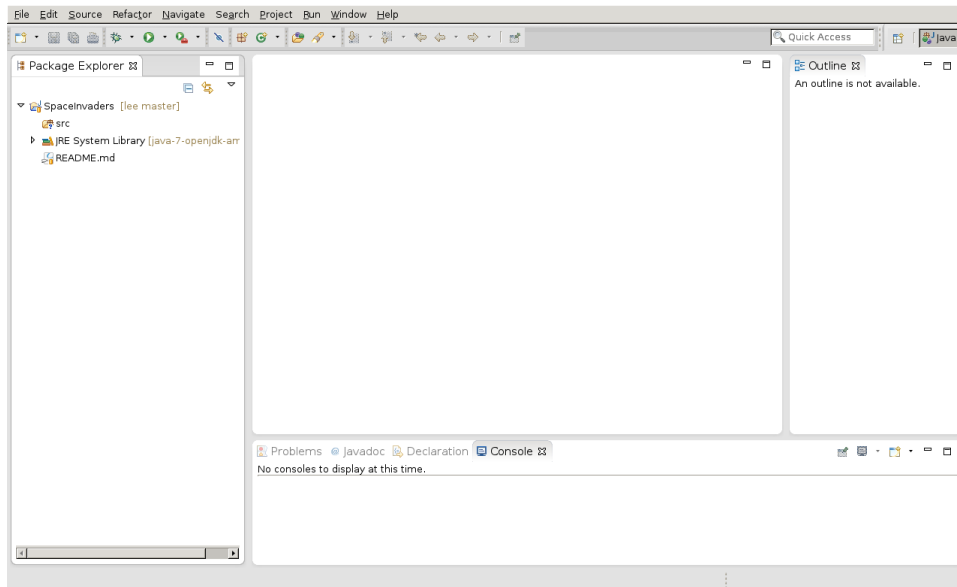


- 1 Wählen Sie das Verzeichnis aus, in welches Sie Ihr Repository geklont haben. (Wenn Sie ein Ordner innerhalb Ihres Repository gemacht haben, um mehrere Projekte im gleichen Repository zu haben, so wählen Sie dieses Verzeichnis aus.)
- 2 Stellen Sie sicher, dass das Projekt hier angezeigt wird und dass Sie es ausgewählt haben.

Nun haben Sie das Projekt ebenfalls im Eclipse.

Erstes Programmieren in Eclipse

Wenn Sie jetzt den **Package Explorer** in Eclipse anschauen, sehen Sie Ihr Projekt. Sie sehen ein Verzeichnis namens **src**. Hier hinein gehört der *Source Code*, d.h. diejenigen Dateien, welche Sie im Verlauf der Entwicklung erstellen und welche Code beinhalten.

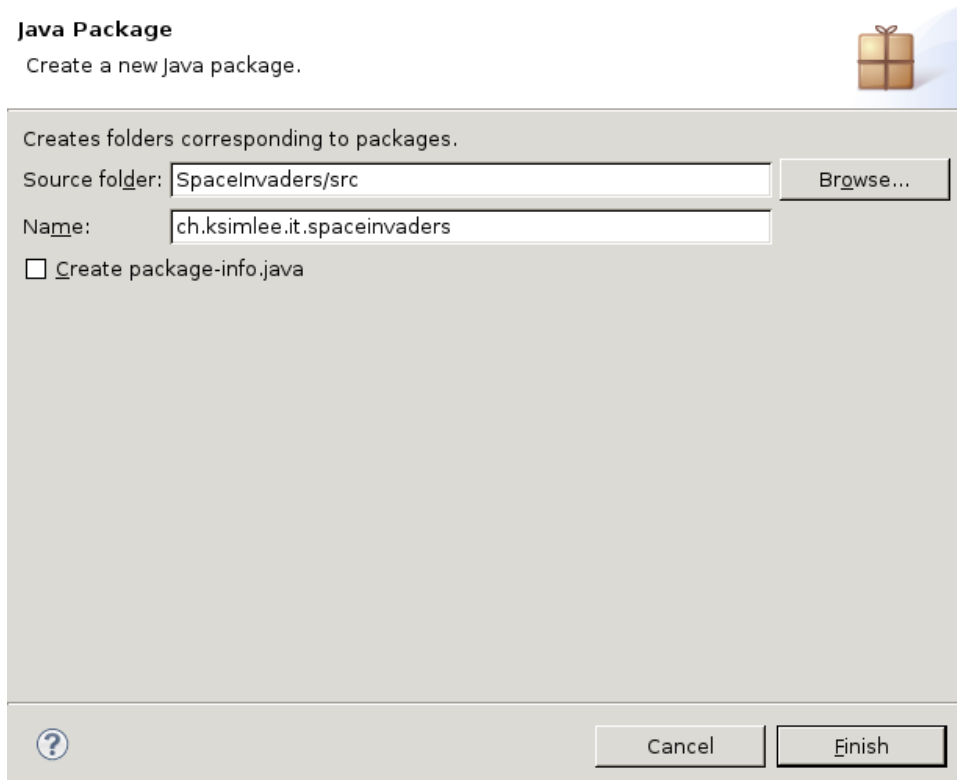


Damit der Source Code besser geordnet ist, und damit nicht alles in einem einzigen Verzeichnis gespeichert ist, gibt es bei Java sogenannte **packages**. Auf dem Dateisystem sind packages einfach Verzeichnisse, d.h. sie können auch verschachtelt werden.

Sie sollten immer innerhalb eines packages arbeiten. Deshalb erstellen wir nun ein erstes package. Bei Java gibt es eine Namenskonvention für packages:

1. Packages bestehen nur aus Kleinbuchstaben.
2. Packages identifizieren den Author und das Projekt, auf ähnliche Weise wie eine URL, aber rückwärts. Siehe nachfolgendes Beispiel.

Um ein package zu erstellen machen wir einen Rechtsklick auf das **src** Verzeichnis und wählen **New > Package**.



Als Namen habe ich hier **ch.ksimlee.it.spaceinvaders** gewählt. Der Punkt erstellt die Verschachtelung, und die einzelnen Abschnitte sind wie folgt gewählt:

ch Das Projekt wurde in der Schweiz erstellt, hat die gleiche Bedeutung wie “.ch” bei Domain Namen.

ksimlee Das Projekt wurde an der KS im Lee erstellt.

it Das Projekt wurde innerhalb der KS im Lee im Bereich Informatik (“IT”) erstellt.

spaceinvaders Der Name des Projektes.

Die Absicht hinter dieser Namensgebung ist, dass der gewählte Name einzigartig ist, und nicht verschiedene Leute den gleichen Namen für verschiedene Projekte verwenden. Das könnte z.B. passieren, wenn ich nur “spaceinvaders” als Projektname gewählt hätte.

Das Package wurde jetzt erstellt, ist aber noch leer.

Eine erste Klasse erstellen

Jetzt, da wir unser Projekt (inklusive package!) vorbereitet haben, können wir eine erste Klasse erstellen. Source Code ist in Java (fast) immer in Klassen (Englisch: Class), denn Klassen sind das zentrale Konzept in Java. Eine Klasse bündelt Funktionalität, welche zusammen gehört.

Wir machen einen Rechtsklick auf unser package, und wählen **New > Class**.

Java Class
Create a new Java class.

Source folder: **Browse...**

Package: **1** **Browse...**

☐ Enclosing type: **Browse...**

Name: **2**

Modifiers: ☒ public ☐ default ☐ private ☐ protected
☐ abstract ☐ final ☐ static

Superclass: **Browse...**

Interfaces: **Add...**
Remove

Which method stubs would you like to create?

☒ `public static void main(String[] args)` **3**

☐ Constructors from superclass

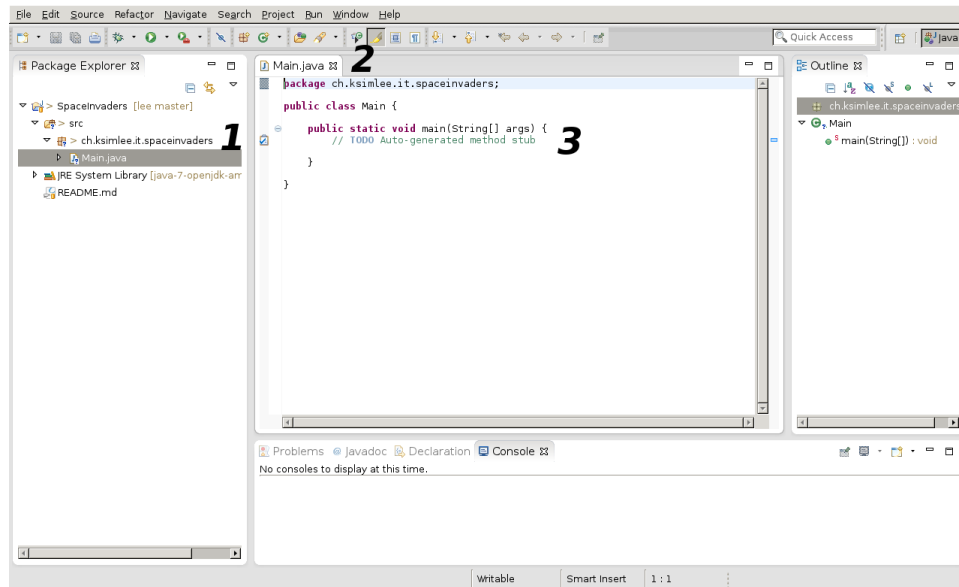
☒ Inherited abstract methods

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

? **Cancel** **Finish**

- 1 Hier sehen Sie, in welchem package die Klasse erstellt wird. Das sollte das package sein, welches wir im letzten Schritt erstellt haben.
- 2 Hier können Sie den Namen der Klasse angeben. Da wir die “Haupt”-Klasse erstellen, also diejenige, welche später unser Programm startet, nenne ich sie **Main**.
Beachte: Klassen Namen sollten mit einem Grossbuchstaben anfangen.
- 3 Jedes Programm hat eine Startfunktion, welche beim Start des Programms ausgeführt wird. Wenn wir diesen Haken setzen, wir die entsprechende Funktionssignatur in der neuen Klasse erstellt.
Beachte: Jedes Programm, d.h. jedes Projekt, braucht nur eine Startfunktion!



1 Die erstellte Klasse namens **Main** ist im Package Explorer zu finden. Sie kann jederzeit durch einen Doppelklick geöffnet werden.

2 Die Klasse **Main** ist momentan geöffnet. Hier kann der source code geändert werden.

3 Diese Funktion (mit dem Namen **main**) wurde von Eclipse für uns erstellt.

Beachte: Die Startfunktion muss immer **main** heissen, unabhängig vom Namen der Klasse!

, wenn wir es ausführen!

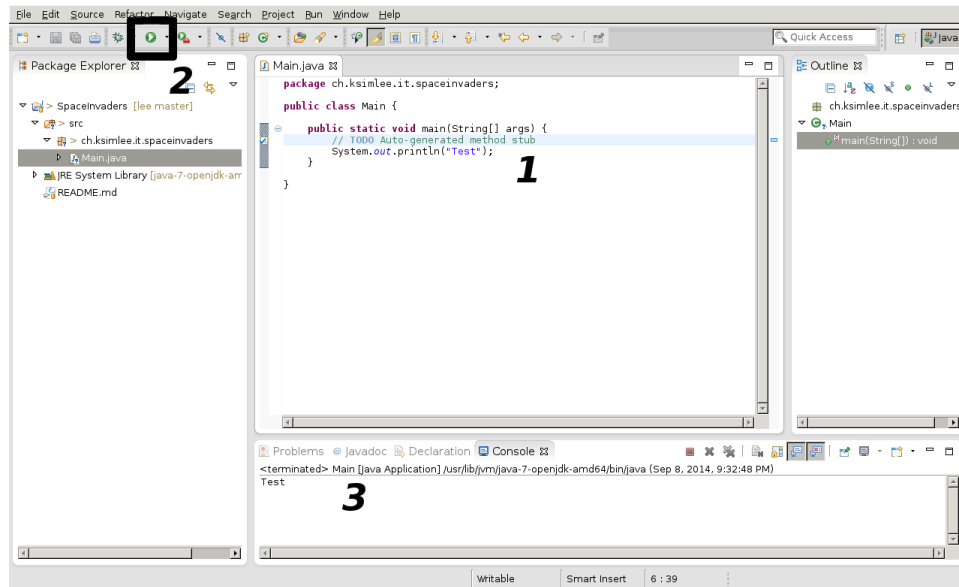
Die erste Klasse wurde nun angelegt, doch da die Funktion noch nichts macht, macht auch unser Programm nichts, wenn wir es ausführen!

Der erste Befehl

Damit unser Programm etwas macht, können wir eine einfache Ausgabe auf der Konsole hinzufügen. Um in Java etwas auf der Konsole auszugeben, gibt es den folgenden “Befehl” (Sie werden später noch verstehen wie der “Befehl” zu Stande kommt):

Listing 1: Dieser Befehl gibt den String “Test” auf der Konsole aus.

```
System.out.println("Test");
```



- 1 Hier haben wir den Befehl zur Ausgabe des Strings "Test" zur **main** Funktion hinzugefügt.
- 2 Mit diesem Button können Sie Ihr Programm ausführen.
- 3 Wenn Sie das Programm ausführen, sehen Sie hier in der Konsole die Ausgabe. Hier sehen Sie den String "Test".

Aufgaben 23.9.

Das Ziel der heutigen Lektion ist es, sich ein wenig mit Swing (der UI Library welche wir verwenden) vertraut zu machen.

1. Im meinem Repository (<https://github.com/stolzda/lee/tree/master/src/ch/ksimlee/it/spaceinvaders>) sind 3 Klassen: fügen sie die Klassen zu ihrem Projekt hinzu. Sie können dies z.B. einfach machen, indem sie zuerst in Eclipse eine Klasse mit dem gleichen Namen anlegen, und dann einfach den Inhalt der Klasse aus dem Browser in das angelegte class file copy pasten. **Beachte: Das Package muss entsprechend umbenannt werden!**
2. Lesen Sie sich die Kommentare in den Klassen durch und versuchen Sie zu verstehen was die einzelnen Sachen machen. Am einfachsten lesen Sie in der Reihenfolge wie der Code ausgeführt wird: Zuerst Main.java, dann Window.java und zuletzt Canvas.java.
3. Ändern Sie den Code so, dass die Zeichnungsfläche einen anderen Hintergrund hat, z.B. eine andere Farbe.
4. Fügen Sie weitere Elemente zur Zeichnungsfläche hinzu, z.B. Rechtecke.
5. Die Zeichnungsfläche (Canvas) wird periodisch neu gezeichnet. Die Anzahl Frames pro Sekunde ist nur eine Vorgabe, wenn der Computer langsam ist, wird er nicht so schnell die Frames zeichnen können. Fügen Sie Code ein, welcher die Anzahl Frames pro Sekunde, welche wirklich gezeichnet wurden zählt (d.h. erhöhen Sie den Zähler am Ende der **paintComponent** Funktion). Geben Sie diese Anzahl aus, entweder auf dem Canvas direkt, oder in der Konsole. Beachte: Es soll jede Sekunde eine Zahl ausgegeben werden!
6. Entwickeln Sie eine Idee für Ihr Spiel!

Aufgaben 30.9.

1. Ich habe die FPS Berechnung in meinem Repository (<https://github.com/stolzda/lee/>) hinzugefügt. **Beachte:** Ich habe drei commits gemacht. Um besser zu sehen, was sich seit dem letzten mal verändert hat, ist es nützlich sich die *Differenz* (**diff**) zum letzten Zustand anzuzeigen. Das können Sie direkt im Browser auf GitHub machen: Klicken sie bei der Repository Übersicht meines Repositories (stolzda/lee) auf “11 commits” damit Sie die Liste der commits sehen. Wenn Sie dann auf einen der Commits klicken, sehen Sie die Differenz zum vorherigen commit.

Schauen Sie die Differenz des commits “Added FPS calculation to Canvas.” an, und bauen Sie mit Hilfe meiner Vorlage die FPS Berechnung in Ihr Spiel ein.

2. Der zweitneueste Commit (“Added game logic and rendering functionality.”) ist wesentlich grösser. Sehen Sie sich auch hier die Differenz an.

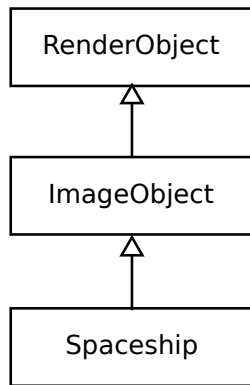
Übernehmen Sie die Änderung in Ihr Projekt. Legen Sie dazu zuerst ein package namens “objects” an, und fügen Sie die drei Klassen, welche ich in meinem package “objects” habe bei Ihnen hinzu. Das geht wieder am einfachsten, wenn Sie die Klassen bei sich zuerst im Package erstellen, und dann aus dem Browser meinen Code reinkopieren, und danach die **package** Zeile innerhalb der Datei korrigieren.

Eine weitere neue Klasse ist **Game**. Fügen Sie auch diese Klasse bei sich im Hauptpackage hinzu.

Wenn Sie sich nun die Differenz anschauen, dann sehen Sie, dass auch an den Klassen **Main**, **Window**, **Canvas** sich etwas geändert hat – jedoch nicht viel. Übernehmen Sie die Änderungen mithilfe der Differenz direkt in Ihre Version der Dateien (**d.h., ersetzen Sie diese drei Dateien bei Ihnen nicht komplett, sondern bauen Sie die Änderungen an den entsprechenden Stellen in der Datei ein!**).

3. Nun haben Sie meine Änderungen übernommen. Was ist konzeptuell neu? Später wollen wir verschiedene Objekte in unserem Spiel haben, welche eine Koordinate und z.B. ein Bild haben. In meinem Code habe ich ein Raumschiff (**Spaceship**) hinzugefügt. Da das Konzept “Objekt, welches an einer Koordinate gezeichnet wird” aber nicht nur für das Raumschiff gilt, sondern auch für alle anderen Objekte, benötigen wir den Code an verschiedenen Stellen. In Objektorientierter Programmierung kann das gut durch **Vererbung** (engl.: inheritance) gelöst werden: Eine Klasse kann eine anderer *erweitern*, d.h., sie übernimmt die Funktionalität der Vaterklasse, und fügt weitere Funktionalität hinzu. In Java kann das durch das keyword **extends** gemacht werden.

Schauen Sie sich die drei Klassen im package **objects** an. Sie sind wie in der folgenden Grafik aufgebaut:



Ein Renderobject ist ein Objekt welches eine Position hat, und gezeichnet werden kann.

Ein ImageObject ist ein RenderObject, jedoch ist festgelegt, dass das Objekt welches gezeichnet wird, ein Bild vom Dateisystem ist.

Ein Spaceship ist ein Objekt, zu welchem wir später noch Spiellogik hinzufügen. Das rendering wird elegant durch Vererbung (d.h. durch ImageObject) gemacht.

Wenn Sie weitere Spielobjekte hinzufügen wollen, können Sie z.B. weitere Subklassen von **ImageObject** (d.h., Klassen welche von **ImageObject** erben) erstellen. Beachten Sie, dass Sie nur eine Klasse, pro *Objektyp* erstellen: So würde z.b. ein Gegner nur eine Klasse "Enemy" benötigen, auch wenn wir später mehrere *Objekte* dieser Klasse erstellen werden (d.h. mehrere Gegner).

4. Schauen Sie sich nun die Klasse **Game** an. Das Game besteht aus einer Schleife (**while(true)**) welche immer ausgeführt wird: Hier drin wird unsere Spiellogik ausgeführt. Zusätzlich hat die Klasse Game eine **Liste**, in welcher alle Objekte welche gezeichnet werden sollen gespeichert sind. (Wenn Sie sich **Canvas** anschauen, sehen Sie, dass diese Liste vom Canvas angeschaut wird, um die Objekte zu zeichnen!)

Die Schleife welche immer ausgeführt wird (in der Funktion **run**) macht momentan noch fast nichts – sie verschiebt einfach das Spaceship nach rechts.

5. Erstellen Sie, analog zu **Spaceship**, Klassen für Ihre Objekte in Ihrem Spiel. Wenn Sie Bilder hinzufügen, müssen diese ins "Hauptverzeichnis" Ihres Projektes, dort wo auch "spaceship.png" ist.
6. Ändern Sie den GameLoop (in der **run** Funktion von **Game**) so, dass Ihre Objekte dort erstellt werden.
 - a) Lassen Sie ein Objekt auf einer Linie hin und her laufen (wie das Spaceship, jedoch nicht mit zurückspringen an den Anfang sondern "umkehren").
 - b) Bewegen Sie ein Objekt in einem Viereck herum.
 - c) Bewegen Sie mehrere Objekte gleichzeitig.
7. Arbeiten Sie an Ihrer Spielidee weiter, oder zeichnen Sie Spielobjekte für Ihr Spiel.

Aufgaben 21.10. – Input Handling

Mein Repository: <https://github.com/stolzda/lee/>

1. Es gibt zwei neue Commits auf meinem Repository. Der erste commit (**Bugfix when having multiple RenderObjects, and only one object was rendered.**) behebt einen Bug welcher verhindert hat, dass man mehrere Objekte im Spiel haben konnte.
⇒ Übernehmen Sie diese Änderungen, damit Sie mehrere Objekte im Spiel haben können.
2. Der zweite commit (**Added input handling. Added example input handling for the spaceship.**) fügt Input handling (d.h., das Verarbeiten von Tastatureingaben) zum Game hinzu.
⇒ Schauen Sie sich den Code an, und übernehmen Sie die entsprechenden Änderungen in Ihren code.
3. Passen Sie den Code so an, dass Sie Ihre eigenen Spielobjekte bewegen können. Sie können die Hilfsfunktion **move(...)** (in der Klasse **RenderObject.java**) verwenden, oder auch eigene Funktionen schreiben.
4. Wenn Sie verschiedene Objekte haben, können Sie diese jeweils mit einer **update(...)** Funktion bewegen – so können Sie z.B. zwei Spieler mit verschiedenen Tasten auf der gleichen Tastatur steuern.
5. Wenn Sie Objekte haben, welche sich unabhängig von der Tastatureingabe bewegen, so können Sie trotzdem eine **update(...)** Funktion schreiben, welche jedoch den **InputHandler** einfach ignoriert.

Aufgaben 28.10. – Collision Detection

Mein Repository: <https://github.com/stolzda/lee/>

Alle Objekte updaten

Letztes mal haben wir im Game Loop auf die einzelnen Objekte **update(...)** aufgerufen, um die Position der Objekte anzupassen. Da wir nicht für jedes Objekt, welches wir zeichnen, einzeln **update** aufrufen wollen, können wir das einfach mit einem Loop über alle Objekte, welche wir gerade zeichnen, machen. (Siehe mein commit *Automatically update all game objects in each iteration of the game loop.*)

Passen Sie ihren Game Loop auch so an, dass Ihre Objekte automatisch in jedem Schritt updated werden.

Grösse zu den Objekten hinzufügen

Im folgenden wollen wir *Kollision* zu unserem Spiel hinzufügen. Um überhaupt feststellen zu können, ob zwei Objekte kollidieren, müssen wir jedem Objekt eine Grösse zuweisen (auch “*bounding box*” genannt). Eine solche bounding box bestimmt den “Umriss” des Objekts, und kann im einfachsten Fall einfach ein Rechteck (darum “box”) sein¹.

Ein Rechteck kann durch eine Position (z.B. die Koordinate der oberen linken Ecke) und die Breite und Höhe eindeutig definiert werden. Da wir die Position bereits bei der Klasse **RenderObject** haben, genügt es wenn wir die Höhe und die Breite auch noch zur Verfügung stellen.

Wir wollen den Unterklassen (den Klassen, welche von **RenderObject** erben, also z.B. die Klasse **ImageObject**) die Möglichkeit geben, selber die Breite und die Höhe des Objektes zu spezifizieren. Darum definieren wir die Funktionen **getWidth()** und **getHeight()** als **abstract**. Das bedeutet, dass die Klassen welche von **RenderObject** diese Funktionen definieren können und auch *müssen*.

In der Klasse **ImageObject** haben wir bereits die Informationen für Breite und Höhe, da Bilder ja als Rechtecke gespeichert werden! Somit können wir diese Information einfach weiterleiten.

Bauen Sie die Funktionalität für die Bereitstellung der Grösse ein. (Vergleiche hierzu mein commit *Added size to ImageObject.*)

(Optional) Bounding boxes darstellen

Wenn Sie die Grössenberechnung eingefügt haben, ist es ein sehr nützliches Feature, wenn Sie die berechnete Bounding box auch direkt zeichnen können. Somit können Sie überprüfen, ob die Werte auch Sinn machen. Ich habe hierzu eine “Zwischen-Funktion” in die Klasse **RenderObject** eingeführt. Der **Canvas** ruft diese Funktion anstelle der **render** Funktion auf – und diese neue Funktion (**renderInternal**) ruft dann sowohl **render** auf, und zeichnet (optional, gesteuert durch ein Flag), die Bounding box. Diese Änderung sind in meinem commit *Added debug functionality to render bounding boxes.*

¹Eine “perfekte” bounding box wäre der genaue Umriss des Objektes, jedoch wäre das rechnerisch sehr viel aufwändiger als einfach ein Rechteck festzulegen – und da das Rechteck meistens genügend gut ist, wird zur Vereinfachung einfach ein Rechteck gemacht.

Auf Kollision prüfen (Vorbereitung)

Jetzt haben wir für jedes Objekt die bounding box – jedoch wird noch nichts damit gemacht. Um Kollisionen feststellen zu können, ist es am einfachsten, wenn wir bei *jedem* Befehl, welcher Objekte bewegt, schauen, ob diese Bewegung zu einer Kollision führt. Das bedeutet, dass wir diese Berechnung direkt in der Funktion **move** machen können.

Damit wir diese Berechnung durchführen können, müssen wir das Set mit allen Objekten auch der **update** Funktion übergeben, damit wir das später der **move** Funktion übergeben können.

Die Vorbereitung für die Kollisionserkennung habe ich im commit *Preparing for collision detection*. gemacht. Zusätzlich habe ich noch das Feld **hasCollision** hinzugefügt, denn wir wollen später auch Objekte haben, die wir nicht für die Kollisionsprüfung verwenden (z.B. Objekte im Hintergrund).

Bereiten Sie die Funktionen analog zu meinem commit vor, dass Sie dann in der Funktion **move** das Set mit allen Objekten haben.

Auf Kollision prüfen

Jetzt müssen Sie die Funktion **move** in der Klasse **RenderObject** so anpassen, dass sie auf Kollision mit anderen Objekten prüft.

Ich habe eine einfache Version implementiert, siehe mein commit *Added algorithm to check for collisions*. Dieser hat eine Hilfsfunktion, welche prüft ob sich zwei Rechtecke überlagern. Mit dieser Funktion können wir eine Kollision feststellen, sofern sich Objekte nicht bewegen. Da Sie sich aber bewegen, benötigen wir einen Algorithmus.

Die Idee des Algorithmus' ist folgende:

1. Gehe einen Schritt in die gewünschte Richtung.
2. Überprüfe ob sich die bounding box mit einem anderen Objekt überlagert \Rightarrow Kollision!
3. Falls es *eine* Kollision gab, gehe einen Schritt zurück und beende die Bewegung.
4. Falls es *keine* Kollision gab, wiederhole diese 4 Schritte solange bis alle Schritte gemacht sind oder es eine Kollision gibt.

Dieser Algorithmus hat einige Schwächen, z.B. ist er nicht besonders effizient, und weiterhin hat er das Problem, dass wir eigentlich, um ganz korrekt zu sein, alle Objekte gleichzeitig verschieben müssten. Für unsere Zwecke ist er aber erst mal gut genug.

Sie können entweder selber einen Kollisionsalgorithmus entwickeln, meine Idee verwenden, oder auch mithilfe meines commits den Algorithmus implementieren, bzw. den Code übernehmen. Wichtig ist, dass Sie eine Kollisionsabfrage haben, welche den Bedürfnissen Ihres Spiels entspricht.

Damit man später von der Kollision weiss, habe ich bei meinem Code die **move** Funktion so ergänzt, dass sie einen **boolean** zurück gibt, welcher uns mitteilt, ob es eine Kollision gab. Je nach Spiel können Sie diesen Rückgabewert auch ändern, bspw. wäre es auch denkbar, dass Sie nicht nur einfach "wahr/falsch" zurückgeben, sondern direkt das Objekt, mit welchem es eine Kollision gab.

Spielelemente hinzufügen

Fügen Sie benötigte Elemente für Ihr Spiel hinzu: Neue Objekte, neue Ereignisse welche passieren wenn man eine Taste drückt, Ereignisse bei Kollisionen. . .