Univerzita Pardubice
Fakulta elektrotechniky a informatiky

Mobilní aplikace

Distanční opora

Ing. Jan Panuš, Ph.D.

1. Proč se učit Kotlin?

Abychom pochopili, kam Kotlin patří a proč se ho vůbec učit, je potřeba uvést přehled historického vývoje programovacích jazyků. Tato kapitola představuje některá témata, která se vám, pokud jste začátečníci, mohou zdát právě teď příliš složitá.

První jazyky se zaměřovaly na hardwarová omezení. S rostoucím výkonem počítačů se novější jazyky posouvají směrem k sofistikovanějšímu programování s důrazem na spolehlivost. Tyto jazyky mohou volit funkce na základě psychologie programování. Každý programovací jazyk je souborem experimentů. Historicky byl návrh programovacích jazyků sledem dohadů a předpokladů o tom, co programátorům přinese vyšší produktivitu. Některé z těchto experimentů selhaly, některé byly mírně úspěšné a některé velmi úspěšné.

Z experimentů se učíme v každém novém jazyce. Některé jazyky řeší problémy, které se ukáží být spíše vedlejšími než zásadními, nebo se změní prostředí (rychlejší procesory, levnější paměť, nové chápání programování a jazyků) a tento problém se stane méně důležitým nebo dokonce nepodstatným. Pokud tyto myšlenky zastarají a jazyk se nevyvíjí, přestane se používat.

Původní programátoři pracovali přímo s čísly reprezentujícími strojové instrukce procesoru. Tento přístup vedl k četným chybám, a proto byl vytvořen jazyk assembler, který nahradil čísla mnemotechnickými kódy - slovy, která si programátoři mohli snadněji zapamatovat a přečíst, spolu s dalšími užitečnými nástroji. Mezi instrukcemi v jazyce assembleru a strojovými instrukcemi však stále existovala shoda jedna ku jedné a programátoři museli každý řádek kódu assembleru napsat. Každý počítačový procesor navíc používal svůj vlastní odlišný jazyk assembleru.

Vývoj programů v assembleru je nesmírně nákladný. Vyšší jazyky pomáhají tento problém řešit tím, že vytvářejí úroveň abstrakce od nízkoúrovňových jazyků assembleru.

1. Překladače a interpreti programovacích jazyků

Jazyk Kotlin je kompilován, nikoli interpretován. Instrukce interpretovaného jazyka jsou prováděny přímo samostatným programem, který se nazývá interpret. Naproti tomu zdrojový kód kompilovaného jazyka je převeden do jiné reprezentace, která běží jako vlastní program, a to buď přímo na hardwarovém procesoru, nebo na virtuálním stroji, který emuluje procesor.

Jazyky jako C, C++, Go a Rust se kompilují do strojového kódu, který běží přímo na hardwarové centrální procesorové jednotce (CPU). Jazyky jako Java a Kotlin se kompilují do tzv. byte kódu, což je formát střední úrovně, který neběží přímo na hardwarovém procesoru, ale na virtuálním stroji, což je program, který instrukce kódu vykonává. Verze Kotlinu pro JVM běží na virtuálním stroji Java (JVM).

Důležitou výhodou virtuálního stroje je přenositelnost. Stejný bajtový kód může běžet na každém počítači, který má virtuální stroj. Virtuální stroje lze optimalizovat pro konkrétní hardware a řešit problémy s rychlostí. JVM obsahuje mnoho let takových optimalizací a byl implementován na mnoha platformách.

Při kompilaci je kód kontrolován kompilátorem, aby byly odhaleny chyby při kompilaci. (IntelliJ IDEA a další vývojová prostředí na tyto chyby upozorňují při zadávání kódu, takže můžete případné problémy rychle odhalit a opravit). Pokud se nevyskytnou žádné chyby při kompilaci, zdrojový kód se zkompiluje do bajtového kódu.

Chybu za běhu nelze odhalit při kompilaci, takže se objeví až při spuštění programu. Chyby runtime se obvykle hůře odhalují a jejich oprava je nákladnější. Staticky typované jazyky, jako je Kotlin, odhalí co

nejvíce chyb v době kompilace, zatímco dynamické jazyky provádějí bezpečnostní kontroly za běhu (některé dynamické jazyky neprovádějí tolik bezpečnostních kontrol, kolik by mohly).

2. Jazyky, které ovlivnily Kotlin

Jazyk Kotlin čerpá své myšlenky a vlastnosti z mnoha jazyků, které byly ovlivněny dřívějšími jazyky. Je užitečné znát trochu historie programovacích jazyků, abychom získali přehled o tom, jak jsme se dostali ke Kotlinu. Jazyky, které jsou zde popsány, jsou vybrány kvůli jejich vlivu na jazyky, které následovaly. Všechny tyto jazyky nakonec inspirovaly návrh jazyka Kotlin, někdy tím, že byly příkladem toho, co se nemá dělat.

FORTRAN: FORmula TRANslation (1957)

Cílem jazyka Fortran, navrženého pro použití vědci a inženýry, bylo usnadnit kódování rovnic. Jemně vyladěné a otestované knihovny Fortranu se používají dodnes, ale obvykle jsou "zabalené" tak, aby je bylo možné volat z jiných jazyků.

LISP: LISt Processor (1958)

Jazyk LISP spíše než specifické aplikace ztělesňoval základní koncepty programování; byl to jazyk počítačových vědců a první funkcionální programovací jazyk (O funkcionálním programování se dozvíte v této knize). Kompromisem za jeho výkon a flexibilitu byla efektivita: Teprve v posledních desetiletích se stroje staly dostatečně rychlými na to, aby se LISP znovu začal používat. Například editor GNU Emacs je celý napsán v jazyce LISP a lze jej pomocí něj rozšiřovat.

ALGOL: ALGOrithmic Language (1958)

Pravděpodobně nejvlivnější z jazyků 50. let, protože zavedl syntaxi, která přetrvala v mnoha následujících jazycích. Například jazyk C a jeho odvozeniny jsou jazyky "podobné ALGOLu".

COBOL: COmmon Business-Oriented Language (1959)

Navržen pro obchodní, finanční a administrativní zpracování dat. Má syntaxi podobnou angličtině a byl zamýšlen jako samodokumentační a velmi čitelný. Ačkoli se tento záměr obecně nepodařil - COBOL je známý chybami, které přinesla chybně umístěná tečka - americké ministerstvo obrany si vynutilo jeho široké přijetí na mainframech a systémy v něm fungují (a vyžadují údržbu) dodnes.

BASIC: univerzální symbolický instrukční kód pro začátečníky (1964)

BASIC byl jedním z prvních pokusů o zpřístupnění programování. Byl sice velmi úspěšný, ale jeho funkce a syntaxe byly omezené, takže byl jen částečně užitečný pro lidi, kteří se potřebovali naučit složitější jazyky. Je to převážně interpretovaný jazyk, což znamená, že k jeho spuštění potřebujete původní kód programu. Přesto bylo v BASICu napsáno mnoho užitečných programů, zejména jako skriptovací jazyk pro produkty Microsoft "Office". BASIC lze dokonce považovat za první "otevřený" programovací jazyk, protože lidé vytvářeli jeho četné varianty.

Simula 67, původní objektově orientovaný jazyk (1967)

Simulace obvykle zahrnuje mnoho "objektů", které na sebe vzájemně působí. Různé objekty mají různé vlastnosti a chování. Jazyky, které v té době existovaly, byly pro simulace nepohodlné, proto byl vyvinut jazyk Simula (další jazyk "podobný ALGOLu"), který poskytoval přímou podporu pro vytváření simulačních objektů. Ukázalo se, že tyto myšlenky jsou užitečné i pro programování pro obecné účely, a tak vznikly objektově orientované (OO) jazyky.

Pascal (1970)

Pascal zvýšil rychlost kompilace omezením jazyka tak, aby mohl být implementován jako jednoprůchodový kompilátor. Jazyk nutil programátora strukturovat kód určitým způsobem a ukládal poněkud nepohodlná a méně čitelná omezení na organizaci programu. S tím, jak se procesory

zrychlovaly, paměť zlevňovala a technologie překladačů se zdokonalovala, se dopad těchto omezení stal příliš nákladným.

Implementace jazyka Pascal, Turbo Pascal od společnosti Borland, zpočátku fungovala na strojích CP/M a poté přešla na raný MS-DOS (předchůdce Windows), později se vyvinula v jazyk Delphi pro Windows. Díky tomu, že se vše ukládalo do paměti, se Turbo Pascal kompiloval bleskovou rychlostí i na velmi málo výkonných strojích, což výrazně zlepšilo zážitek z programování. Jeho tvůrce Anders Hejlsberg později navrhl jazyk C# i TypeScript.

Niklaus Wirth, vynálezce jazyka Pascal, vytvořil další jazyky: Modula, Modula-2 a Oberon. Jak už název napovídá, Modula se zaměřovala na rozdělení programů do modulů pro lepší organizaci a rychlejší kompilaci. Většina moderních jazyků podporuje oddělenou kompilaci a nějakou formu modulového systému.

C (1972)

Navzdory rostoucímu počtu jazyků vyšší úrovně programátoři stále psali v assembleru. Často se tomu říká systémové programování, protože se provádí na úrovni operačního systému, ale patří sem i vestavěné programování pro vyhrazená fyzická zařízení. To je nejen namáhavé a nákladné (Bruce začal svou kariéru psaním assembleru pro vestavné systémy), ale není to ani přenositelné - assembler může běžet pouze na procesoru, pro který je napsán. Jazyk C byl navržen jako "vysokoúrovňový jazyk assembleru", který je stále dostatečně blízký hardwaru, takže jen zřídkakdy je třeba psát assembler. Důležitější je, že program v jazyce C běží na jakémkoli procesoru s kompilátorem jazyka C. Jazyk C oddělil program od procesoru, což vyřešilo obrovský a nákladný problém. Výsledkem bylo, že bývalí programátoři v assembleru mohli být mnohem produktivnější v jazyce C. Jazyk C byl tak efektivní, že se ho nejnovější jazyky (zejména Go a Rust) stále pokoušejí uzurpovat pro systémové programování.

Smalltalk (1972)

Smalltalk byl od počátku navržen jako čistě objektově orientovaný a významně posunul teorii OO a jazyků kupředu tím, že se stal platformou pro experimentování a ukázku rychlého vývoje aplikací. Byl však vytvořen v době, kdy jazyky byly ještě proprietární, a vstupní cena systému Smalltalk se mohla pohybovat v řádech tisíců. Byl interpretovaný, takže ke spouštění programů bylo potřeba prostředí Smalltalku. Open-source implementace jazyka Smalltalk se objevily až poté, co se svět programování posunul dál. Programátoři Smalltalku přispěli velkými poznatky, které byly přínosem pro pozdější OO jazyky jako C++ a Java.

C++: C++: Lepší C s objekty (1983)

Bjarne Stroustrup vytvořil C++, protože chtěl lepší C a chtěl podporu pro objektově orientované konstrukce, s nimiž se setkal při používání Simuly-67. Na začátku roku 1983 se objevil nový jazyk C++. Bruce byl prvních osm let členem výboru pro standardy C++ a napsal tři knihy o C++ včetně knihy Thinking in C++.

Zpětná kompatibilita s jazykem C byla základním principem návrhu jazyka C++, takže kód jazyka C lze kompilovat v jazyce C++ prakticky beze změn. To umožnilo snadnou migraci - programátoři mohli nadále programovat v jazyce C, využívat výhod jazyka C++ a pomalu experimentovat s funkcemi jazyka C++ a přitom zůstat produktivní. Většinu kritiky jazyka C++ lze vysledovat k omezení zpětné kompatibility s jazykem C.

Jedním z problémů jazyka C byla otázka správy paměti. Programátor musí nejprve získat paměť, pak spustit operaci využívající tuto paměť a poté paměť uvolnit. Zapomenutí uvolnit paměť se nazývá únik paměti a může vést k vyčerpání dostupné paměti a pádu procesu. Původní verze jazyka C++

přinesla v této oblasti několik inovací, spolu s konstruktory zajišťujícími správnou inicializaci. Pozdější verze jazyka přinesly v oblasti správy paměti významná vylepšení.

Python: (1990)

Tvůrce jazyka Python Guido Van Rossum vytvořil tento jazyk na základě své inspirace "programováním pro každého". Díky jeho péči o komunitu jazyka Python má tato komunita pověst nejpřátelštější a nejpodpůrnější komunity ve světě programování. Python byl jedním z prvních jazyků s otevřeným zdrojovým kódem, což vedlo k implementaci prakticky na všech platformách včetně vestavěných systémů a strojového učení. Díky své dynamičnosti a snadnému použití je ideální pro automatizaci malých, opakujících se úloh, ale jeho funkce podporují i tvorbu rozsáhlých, komplexních programů.

Python je skutečný "grass-roots" jazyk; nikdy jej nepropagovala žádná společnost a přístup jeho fanoušků byl takový, že se nikdy nesnažili jazyk prosazovat, ale jednoduše pomáhali každému, kdo se jej chtěl naučit. Jazyk se neustále zdokonaluje a v posledních letech jeho popularita prudce vzrostla.

Python byl možná prvním mainstreamovým jazykem, který kombinoval funkcionální a OO programování. Předcházel Javu automatickou správou paměti pomocí garbage collection (obvykle nikdy nemusíte sami alokovat nebo uvolňovat paměť) a možností spouštět programy na více platformách.

Haskell: (1990)

Jazyk Haskell, inspirovaný proprietárním jazykem Miranda (1985), byl vytvořen jako otevřený standard pro výzkum čistého funkcionálního programování, ačkoli byl používán i pro produkty. Syntaxe a myšlenky jazyka Haskell ovlivnily řadu pozdějších jazyků včetně jazyka Kotlin.

Java: (1995): Virtuální stroje a garbage collection (1995)

James Gosling a jeho tým dostali za úkol napsat kód pro televizní set-top box. Rozhodli se, že se jim nelíbí jazyk C++, a místo krabičky vytvořili jazyk Java. Společnost Sun Microsystems vyvinula na tento svobodný jazyk (v té době ještě nový nápad) obrovskou marketingovou podporu, aby se pokusila ovládnout vznikající internetové prostředí.

Toto domnělé časové okno pro ovládnutí internetu vyvinulo velký tlak na návrh jazyka Java, což vedlo ke značnému množství nedostatků (kniha *Myšlení v Javě* tyto nedostatky osvětluje, aby byli čtenáři připraveni se s nimi vyrovnat). Brian Goetz ze společnosti Oracle, současný vedoucí vývojář jazyka Java, provedl v Javě pozoruhodná a překvapivá zlepšení navzdory omezením, která zdědil. Přestože Java byla pozoruhodně úspěšná, důležitým cílem návrhu Kotlinu je odstranit nedostatky Javy, aby programátoři mohli být produktivnější.

Úspěch Javy přinesly dva inovativní prvky: virtuální stroj a garbage collection. Ty byly k dispozici i v jiných jazycích - například LISP, Smalltalk a Python mají garbage collection a UCSD Pascal běžel na virtuálním stroji - ale nikdy nebyly považovány za praktické pro běžné jazyky. Java to změnila, a tím výrazně zvýšila produktivitu programátorů.

Virtuální stroj je mezivrstva mezi jazykem a hardwarem. Jazyk nemusí generovat strojový kód pro konkrétní procesor; stačí, když vygeneruje mezikód (bytecode), který běží na virtuálním stroji. Virtuální stroje vyžadují výpočetní výkon a před nástupem Javy byly považovány za nepraktické. Virtuální stroj Java (Java Virtual Machine, JVM) dal vzniknout sloganu Javy "jednou napiš, spustíš všude". Kromě toho lze díky zaměření na JVM snadněji vyvíjet další jazyky; příkladem je Groovy, skriptovací jazyk podobný Javě, a Clojure, verze jazyka LISP.

Garbage collection řeší problém, kdy se zapomene uvolnit paměť nebo kdy je obtížné zjistit, kdy už se část paměti nepoužívá. Projekty byly kvůli únikům paměti výrazně zpožděny nebo dokonce zrušeny. Ačkoli se garbage collection objevuje v některých předchozích jazycích, mělo se za to, že přináší nepřijatelnou režii, dokud Java neprokázala jeho praktičnost.

JavaScript: (1995)

Původní webový prohlížeč jednoduše kopíroval a zobrazoval stránky z webového serveru. Webové prohlížeče se rozšířily a staly se novou programovací platformou, která potřebovala jazykovou podporu. Tímto jazykem chtěla být Java, ale byla pro tuto práci příliš nešikovná. JavaScript vznikl jako LiveScript a byl zabudován do NetScape Navigatoru, jednoho z prvních webových prohlížečů. Přejmenování na JavaScript byl marketingový tah společnosti NetScape, protože tento jazyk se Javě podobá jen matně.

S rozmachem webu se JavaScript stal nesmírně důležitým. Chování JavaScriptu však bylo tak nepředvídatelné, že Douglas Crockford napsal knihu s jazykolamným názvem JavaScript, dobré části, kde ukázal všechny problémy tohoto jazyka, aby se jim programátoři mohli vyhnout. Následná vylepšení provedená výborem ECMAScript způsobila, že JavaScript je pro původního programátora v JavaScriptu k nepoznání. Nyní je považován za stabilní a vyspělý jazyk.

Webové assembler (WASM) byl odvozen z JavaScriptu, aby se stal jakýmsi bytekódem pro webové prohlížeče. Často běží mnohem rychleji než JavaScript a může být generován jinými jazyky. V době psaní tohoto článku pracuje tým Kotlinu na přidání WASM jako cílového jazyka.

C#: Java pro .NET (2000)

Jazyk C# byl navržen tak, aby poskytoval některé důležité schopnosti jazyka Java na platformě .NET (Windows) a zároveň osvobodil návrháře od omezení následovat jazyk Java. Výsledek zahrnoval četná vylepšení oproti Javě. V jazyce C# byl například vyvinut koncept rozšiřujících funkcí, které jsou hojně využívány v jazyce Kotlin. Jazyk C# se také stal výrazně funkčnějším než Java. Mnoho funkcí jazyka C# zjevně ovlivnilo návrh jazyka Kotlin.

Scala: SCALAble (2003)

Martin Odersky vytvořil jazyk Scala pro běh na virtuálním stroji Java: Aby navázala na práci vykonanou na JVM, spolupracovala s programy v Javě a možná i s myšlenkou, že by mohla Javu vytlačit. Jako výzkumník používal Odersky a jeho tým Scalu jako platformu pro experimentování s vlastnostmi jazyka, zejména s těmi, které nejsou obsaženy v Javě.

Tyto experimenty byly poučné a řada z nich si našla cestu do jazyka Kotlin, obvykle v upravené podobě. Například možnost nadefinovat operátory jako + pro použití ve speciálních případech se nazývá přetěžování operátorů. To bylo zahrnuto v jazyce C++, ale ne v Javě. Scala přetěžování operátorů přidala, ale také umožňuje vymýšlet nové operátory kombinací libovolných posloupností znaků. To často vytváří matoucí kód. Omezená forma přetěžování operátorů je zahrnuta v jazyce Kotlin, ale přetěžovat můžete pouze operátory, které již existují.

Scala je také objektově-funkční hybrid, podobně jako Python, ale se zaměřením na čisté funkce a striktní objekty. To pomohlo inspirovat Kotlin k rozhodnutí být také objektově-funkčním hybridem.

Stejně jako Scala běží Kotlin na JVM, ale s Javou spolupracuje mnohem snadněji než Scala. Kromě toho se Kotlin zaměřuje na JavaScript, operační systém Android a generuje nativní kód pro další platformy.

Groovy: Dynamický jazyk JVM (2007)

Dynamické jazyky jsou přitažlivé, protože jsou interaktivnější a stručnější než statické jazyky. Existuje řada pokusů o vytvoření dynamičtějšího programování v JVM, včetně Jythonu (Python) a Clojure (dialekt jazyka Lisp). Groovy byl první, který dosáhl širokého přijetí.

Na první pohled se Groovy jeví jako vyčištěná verze Javy, která vytváří příjemnější programovací prostředí. Většina kódu v jazyce Java poběží v Groovy beze změny, takže programátoři v jazyce Java mohou být rychle produktivní a později se naučit sofistikovanější funkce, které poskytují znatelné vylepšení programování oproti jazyku Java.

Operátory Kotlinu ?. a ?:, které řeší problém prázdnoty (nullable0, se poprvé objevily v jazyce Groovy.

Existuje řada funkcí jazyka Groovy, které jsou rozpoznatelné v jazyce Kotlin. Některé z těchto funkcí se objevují i v jiných jazycích, což pravděpodobně více tlačilo na jejich začlenění do jazyka Kotlin.

Kotlin (Představen v roce 2011, verze 1.0: 2016)

Stejně jako jazyk C++ měl být zpočátku "lepším jazykem C", Kotlin se zpočátku orientoval na to, aby byl "lepší Javou". Od té doby se vyvinul výrazně nad rámec tohoto cíle.

Kotlin pragmaticky vybírá pouze nejúspěšnější a nejužitečnější funkce z jiných programovacích jazyků - poté, co byly tyto funkce otestovány v praxi a ukázaly se jako obzvláště cenné.

Pokud tedy přicházíte z jiného jazyka, můžete v Kotlinu rozpoznat některé vlastnosti tohoto jazyka. To je záměrné: Kotlin maximalizuje produktivitu tím, že využívá osvědčené koncepty.

Čitelnost

Čitelnost je hlavním cílem při návrhu jazyka. Syntaxe jazyka Kotlin je stručná - pro většinu scénářů nevyžaduje žádný obřad, ale přesto dokáže vyjádřit složité myšlenky.

Nástroje

Jazyk Kotlin pochází od společnosti JetBrains, která se specializuje na vývojářské nástroje. Má prvotřídní podporu nástrojů a mnoho funkcí jazyka bylo navrženo s ohledem na nástroje.

Víceparadigmový

Jazyk Kotlin podporuje více programovacích paradigmat, která jsou v této knize jemně představena:

- Imperativní programování
- Funkcionální programování
- Objektově orientované programování

Multiplatformní

Zdrojový kód jazyka Kotlin lze zkompilovat pro různé cílové platformy:

- JVM. Zdrojový kód se zkompiluje do bytového kódu JVM (.class soubory), který lze následně spustit na libovolném virtuálním stroji Java (JVM).
- Android. Android vlastní běhové prostředí s názvem ART (předchůdce se jmenoval Dalvik). Zdrojový kód jazyka Kotlin se kompiluje do spustitelného formátu Dalvik (.dex soubory).
- JavaScriptu, který se spouští uvnitř webového prohlížeče.
- Nativní binární soubory generováním strojového kódu pro konkrétní platformy a procesory.

Interoperabilita Javy bez námahy

Aby byl jazyk C++ "lepším jazykem C", musí být zpětně kompatibilní se syntaxí jazyka C, ale jazyk Kotlin nemusí být zpětně kompatibilní se syntaxí jazyka Java - musí pouze spolupracovat s JVM. To

návrhářům jazyka Kotlin umožňuje vytvořit mnohem čistší a výkonnější syntaxi bez vizuálního šumu a komplikací, které zahlcují Javu.

Aby byl Kotlin "lepší Javou", musí být zkušenost s jeho vyzkoušením příjemná a bezproblémová, takže Kotlin umožňuje bezproblémovou integraci se stávajícími projekty v Javě. Můžete napsat malý kousek funkce jazyka Kotlin a vsunout jej mezi stávající kód jazyka Java. Kód v Javě ani nepozná, že tam kód Kotlin je - prostě vypadá jako další kód v Javě.

Firmy často zkoumají nový jazyk tak, že vytvoří samostatný program s tímto jazykem. V ideálním případě je tento program přínosný, ale není nezbytný, takže v případě neúspěchu projektu jej lze ukončit s minimálními škodami. Ne každá společnost chce vynakládat takové prostředky, které jsou pro tento typ experimentování nezbytné. Vzhledem k tomu, že jazyk Kotlin se bezproblémově integruje do stávajícího systému Java (a těží z testů tohoto systému), je velmi levné nebo dokonce bezplatné vyzkoušet jazyk Kotlin a zjistit, zda se hodí.

Společnost JetBrains, která Kotlin vytváří, navíc poskytuje IntelliJ IDEA ve verzi "Community" (zdarma), která obsahuje podporu pro Javu i Kotlin spolu s možností snadné integrace obou jazyků. Má dokonce nástroj, který vezme kód v jazyce Java a (většinou) jej přepíše do jazyka Kotlin.

Reprezentace prázdnoty

Zvláště přínosnou vlastností jazyka Kotlin je jeho řešení náročného programátorského problému.

Co uděláte, když vám někdo podá slovník a požádá vás, abyste vyhledali slovo, které neexistuje? Mohli byste zaručit výsledky tím, že si vymyslíte definice neznámých slov. Užitečnějším přístupem je prostě říct: "Pro toto slovo neexistuje žádná definice." To se vám může stát. To ukazuje na významný problém v programování: Jak označíte "žádná hodnota" pro část úložiště, která není inicializovaná, nebo pro výsledek operace?

Nulovou referenci (null reference) vymyslel v roce 1965 pro jazyk ALGOL Tony Hoare, který ji později nazval "mým omylem za miliardu dolarů". Jedním z problémů bylo, že byla příliš jednoduchá - někdy nestačí, že je místnost prázdná; můžete například potřebovat vědět, proč je prázdná. To vede k druhému problému: implementaci. Kvůli efektivitě šlo typicky jen o speciální hodnotu, která se vešla do malého množství paměti, a co bylo lepší než paměť, která již byla pro tuto informaci alokována?

Původní jazyk C paměť automaticky neinicializoval, což způsobovalo řadu problémů. Jazyk C++ situaci zlepšil tím, že nově alokované úložiště nastavil na všechny nuly. Pokud tedy není číselná hodnota inicializována, je to prostě číselná nula. To se nezdálo tak špatné, ale umožňovalo to neinicializovaným hodnotám nenápadně proklouznout (novější kompilátory jazyků C a C++ na to často upozorňují). A co hůř, pokud byl kus paměti ukazatelem - používal se k označení ("ukázání na") jiný kus paměti - nulový ukazatel by ukazoval na nulové místo v paměti, což téměř jistě není to, co chcete.

Java zabraňuje přístupu k neinicializovaným hodnotám tím, že takové chyby hlásí za běhu. Ačkoli se tím odhalí neinicializované hodnoty, problém to neřeší, protože jediný způsob, jak můžete ověřit, že váš program nespadne, je jeho spuštění. V kódu Javy se vyskytují roje těchto typů chyb a programátoři ztrácejí obrovské množství času jejich hledáním.

Kotlin tento problém řeší tím, že zabraňuje operacím, které by mohly způsobit chybu null, již při kompilaci, tedy ještě před spuštěním programu. To je jediná a nejvíce oslavovaná vlastnost programátorů Javy, kteří si Kotlin osvojili. Tato jediná funkce dokáže minimalizovat nebo eliminovat nulové chyby Javy.

Vytvořeno v rámci projektu: **DANTE**, reg. č. NPO_UPCE_MSMT-16591/2022

Financováno Evropskou unií

NextGenerationEU

Národní plán obnovy

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Objekty

Objekty ukládají data pomocí *vlastností* (`vals` a `vars`) a provádějí s nimi operace pomocí funkcí.

Některé definice:

- *Třída*: Definuje vlastnosti a funkce pro nový datový typ. Třídy se také nazývají *uživatelsky definované typy*.
- *Člen*: Člen: buď vlastnost, nebo funkce třídy.
- *Funkce člena*: Funkce, která pracuje pouze s určitou třídou objektu.
- *Vytvoření objektu*: Vytvoření `val` nebo `var` třídy. Nazývá se také *vytvoření instance* této třídy.

Protože třídy definují *stav* a *chování,* můžeme se na instance vestavěných typů, jako je `Double` nebo `Boolean`, odkazovat jako na objekty.

Vezměme si třídu `IntRange` jazyka Kotlin:

```
// ObjectsEverywhere/IntRanges.kt

fun main() {
  hodnota r1 = IntRange(0, 10)
  hodnota r2 = IntRange(5, 7)
  println(r1)
  println(r2)
}
/* Výstup:
0..10
5..7
*/
```

Vytvoříme dva objekty (instance) *třídy* `IntRange`. Každý objekt má v paměti svůj vlastní úložný prostor. `IntRange` je třída, ale konkrétní rozsah `r1` od 0 do 10 je objekt, který je odlišný od rozsahu `r2`.

Pro objekt `IntRange` je k dispozici řada operací. Některé z nich jsou jednoduché, jako například `funkce sum()`, jiné vyžadují více znalostí, než je budete moci použít. Pokud se pokusíte zavolat operaci, která vyžaduje argumenty, IDE se vás na ně zeptá.

Chcete-li se dozvědět více o konkrétní členské funkci, vyhledejte ji v dokumentaci jazyka Kotlin. Všimněte si ikony lupy v pravém horním rohu stránky. Klikněte na ni a do vyhledávacího pole zadejte `IntRange.` Ve výsledném vyhledávání klikněte na `kotlin.ranges > IntRange.` Zobrazí se dokumentace třídy `IntRange.` Můžete si prostudovat všechny členské funkce - *aplikační programové rozhraní* (API) - této třídy. Ačkoli většině z nich v tuto chvíli nebudete rozumět, je užitečné si osvojit vyhledávání v dokumentaci Kotlinu.

`IntRange` je druh objektu a charakteristickým znakem objektu je, že s ním lze provádět operace. Místo "provedení operace" říkáme *volání členské funkce*. Volání členské funkce objektu začínáme identifikátorem objektu, pak tečkou a pak názvem operace:

```
// ObjectsEverywhere/RangeSum.kt

fun main() {
  val r = IntRange(0, 10)
  println(r.sum())
}
/* Výstup:
55
*/
```

Protože `funkce sum()` je členskou funkcí definovanou pro `IntRange`, voláte ji příkazem `r.sum()`. Tím se sečtou všechna čísla v tomto `IntRange`.

Dřívější objektově orientované jazyky používaly výraz "odeslání zprávy" pro volání členské funkce objektu. Někdy se s touto terminologií setkáte i dnes.

Třídy mohou mít mnoho operací (členských funkcí). Třídy lze snadno prozkoumat pomocí IDE (integrovaného vývojového prostředí), které obsahuje funkci zvanou *doplňování kódu*. Například pokud v prostředí IntelliJ IDEA zadáte za identifikátor objektu `.s,` zobrazí se všechny členy tohoto objektu začínající na `s`:

Dokončení kódu

Zkuste použít doplňování kódu pro jiné objekty. Můžete například obrátit `řetězec` nebo převést všechny znaky na malá písmena:

```
// ObjectsEverywhere/Strings.kt

fun main() {
  val s = 'AbcD'
  println(s.reversed())
  println(s.toLowerCase())
}
/* Výstup:
DcbA
abcd
*/
```

`Řetězec` můžete snadno převést na celé číslo a zpět:

```
// ObjectsEverywhere/Conversion.kt

fun main() {
  val s = '123'
  println(s.toInt())
  hodnota i = 123
  println(i.toString())
}
/* Výstup:
123
```

```
123
*/
```

Později v knize probereme strategie pro řešení situací, kdy `řetězec`, který chcete převést, nepředstavuje správnou hodnotu celého čísla.

Můžete také převádět z jednoho číselného typu na jiný. Aby nedošlo k záměně, jsou konverze mezi číselnými typy explicitní. Například `Int i` převedete na `Long` voláním `i.toLong()` nebo na `Double` voláním `i.toDouble()`:

```
// ObjectsEverywhere/NumberConversions.kt

fun fraction(numerator: Long, denom: Long) =
  numerator.toDouble() / denom

fun main() {
  hodnota num = 1
  hodnota den = 2
  val f = fraction(num.toLong(), den.toLong())
  println(f)
}
/* Výstup:
0.5
*/
```

Dobře definované třídy jsou pro programátora snadno pochopitelné a vytvářejí kód, který je snadno čitelný.

Vytváření tříd

Můžete používat nejen předdefinované typy, jako jsou `IntRange` a `String`, ale také vytvářet vlastní typy objektů.

Vytváření nových typů skutečně tvoří většinu činností v objektově orientovaném programování. Nové typy se vytvářejí definováním *tříd*.

Objekt je součástí řešení problému, který se snažíte vyřešit. Začněte o objektech přemýšlet jako o vyjádření pojmů. Pokud ve svém problému objevíte nějakou "věc", představte ji v prvním přiblížení jako objekt ve svém řešení.

Předpokládejme, že chcete vytvořit program pro správu zvířat v zoologické zahradě. Má smysl rozdělit různé typy zvířat do kategorií podle toho, jak se chovají, jaké mají potřeby, s jakými zvířaty vycházejí a s jakými se perou. Vše, co je na daném druhu zvířete odlišné, je zachyceno v klasifikaci předmětu tohoto zvířete. Kotlin používá klíčové slovo `class` k vytvoření nového typu objektu:

```
// CreatingClasses/Animals.kt

// Vytvoření několika tříd:
třída Žirafa
třída Bear
třída Hippo

fun main() {
  // Vytvoření některých objektů:
  val g1 = Žirafa()
  val g2 = Žirafa()
  val b = Bear()
  hodnota h = Hippo()

  // Každý objekt() je jedinečný:
  println(g1)
  println(g2)
  println(h)
  println(b)
}
/* Ukázka výstupu:
Žirafa@28d93b30
```

```
Žirafa@1b6d3586
Hippo@4554617c
Medvěd@74a14482
*/
```

Chcete-li definovat třídu, začněte klíčovým slovem `class`, za kterým následuje identifikátor nové třídy. Název třídy musí začínat písmenem (A-Z, velká nebo malá písmena), ale může obsahovat i čísla a podtržítka. Podle konvence píšeme první písmeno názvu třídy s velkým písmenem a první písmeno všech `vals` a `vars` s malým písmenem.

`Animals.kt` začíná definicí tří nových tříd a poté vytvoří čtyři objekty (nazývané také *instance)* těchto tříd.

`Žirafa` je třída, ale konkrétní pětiletý žirafí samec, který žije v Botswaně, je *objekt*. Každý objekt se liší od všech ostatních, proto jim dáváme jména jako `g1` a `g2`.

Všimněte si poněkud záhadného výstupu posledních čtyř řádků. Část před znakem `@` je název třídy a číslo za `znakem @` je adresa, na které se objekt nachází v paměti počítače. Ano, je to číslo, i když obsahuje několik písmen - říká se tomu "hexadecimální zápis". Každý objekt ve vašem programu má svou vlastní jedinečnou adresu.

Zde definované třídy (`Žirafa`, `Medvěd` a `Hroch`) jsou co nejjednodušší: celá definice třídy je na jediném řádku. Složitější třídy používají kudrnaté závorky (`{` a `}`) k vytvoření *těla třídy, které* obsahuje vlastnosti a chování dané třídy.

Funkce definovaná v rámci třídy patří do této třídy. V Kotlinu tyto funkce nazýváme *členské funkce* třídy. Některé objektově orientované jazyky, jako je Java, se jim rozhodly říkat *metody,* což je termín pocházející z raných objektově orientovaných jazyků, jako je Smalltalk. Aby zdůraznili funkční povahu jazyka Kotlin, rozhodli se návrháři upustit od termínu *metoda,* protože pro některé začátečníky bylo toto rozlišení matoucí. Místo toho se v celém jazyce používá termín *funkce*.

Pokud je to jednoznačné, řekneme prostě "funkce". Pokud musíme rozlišovat:

- *Členské* funkce patří do třídy.
- Funkce *nejvyšší úrovně* existují samy o sobě a nejsou součástí třídy.

Zde patří funkce `bark()` do třídy `Dog`:

```
// CreatingClasses/Dog.kt

třída Dog {
  fun bark() = 'yip!'
}

fun main() {
  val dog = Dog()
}
```

V příkazu `main()` vytvoříme objekt `Dog` a přiřadíme ho do `val dog`. Kotlin vydá varování, protože nikdy nepoužíváme `dog`.

Členské funkce se volají (*vyvolávají*) pomocí názvu objektu, za nímž následuje znak `.` (tečka/tečka), po němž následuje název funkce a seznam parametrů. Zde voláme funkci `meow()` a zobrazujeme výsledek:

```
// CreatingClasses/Cat.kt

třída Cat {
  fun meow() = 'mrrrow!'
}

fun main() {
  val cat = Cat()
  // Volání 'meow()' pro 'cat':
  val m1 = cat.meow()
  println(m1)
}
/* Výstup:
```

```
mrrrow!
*/
```

Členská funkce působí na konkrétní instanci třídy. Když voláte funkci `meow()`, musíte ji volat s objektem. Během volání může `funkce meow()` přistupovat k dalším členům tohoto objektu.

Při volání členské funkce Kotlin sleduje objekt zájmu tím, že mu tiše předá referenci na tento objekt. Tento odkaz je dostupný uvnitř členské funkce pomocí klíčového slova `this`.

Členské funkce mají speciální přístup k ostatním prvkům v rámci třídy, a to jednoduše tak, že tyto prvky pojmenují. Přístup k těmto prvkům můžete také explicitně *kvalifikovat* pomocí `této funkce`. Zde `exercise()` volá `speak()` s kvalifikací i bez ní:

```
// CreatingClasses/Hamster.kt

třída Hamster {
  fun speak() = 'Squeak! '
  fun exercise() =
    this.speak() + // Kvalifikováno pomocí 'this'
      speak() + // Bez 'this'
      "Běh na kole
}

fun main() {
  val hamster = Křeček()
  println(hamster.exercise())
}
/* Výstup:
Squeak! Squeak! Běh na kole
*/
```

V příkazu `exercise()` voláme `speak()` nejprve s explicitním `this` a poté vynecháme kvalifikaci.

Někdy se setkáte s kódem obsahujícím zbytečně explicitní `this`. Takový kód často pochází od programátorů, kteří znají jiný jazyk, kde je `to` buď vyžadováno, nebo je `to` součástí jeho stylu. Zbytečné použití funkce je pro čtenáře matoucí a stráví čas tím, že se snaží zjistit, proč to děláte. Doporučujeme se zbytečnému používání vyhnout.

Mimo třídu musíte říci `hamster.exercise()` a `hamster.speak()`.

Vlastnosti

Vlastnost je `proměnná` nebo `hodnota`, která je součástí třídy.

Definování vlastnosti *udržuje stav v* rámci třídy. Udržování stavu je hlavním motivačním důvodem pro vytvoření třídy namísto pouhého zápisu jedné nebo více samostatných funkcí.

Vlastnost `var` lze znovu přiřadit, zatímco vlastnost `val` nikoli. Každý objekt má své vlastní úložiště pro vlastnosti:

```
// Vlastnosti/Cup.kt

třída Cup {
  var percentFull = 0
}

fun main() {
  hodnota c1 = Cup()
  c1.percentFull = 50
  hodnota c2 = Cup()
  c2.percentFull = 100

  println(c1.percentFull)
  println(c2.percentFull)
}
/* Výstup:
```

```
50
100
*/
```

Definice `var` nebo `val` uvnitř třídy vypadá stejně jako definice uvnitř funkce. `Var` nebo `val` se však stává *součástí* této třídy a musíte se na něj odkazovat tak, že uvedete jeho objekt pomocí *tečkové notace,* přičemž mezi objekt a název vlastnosti umístíte tečku. Vidíte, že tečková notace se používá pro každý odkaz na `procentFull`.

Vlastnost `percentFull` představuje stav příslušného objektu `Cup`. `c1.percentFull` a `c2.percentFull` obsahují různé hodnoty, což ukazuje, že každý objekt má své vlastní úložiště.

Členská funkce může odkazovat na vlastnost v rámci svého objektu bez použití tečkové notace (tj. bez *upřesnění)*:

```
// Vlastnosti/Cup2.kt

třída Cup2 {
  var percentFull = 0
  hodnota max = 100
  fun add(increase: Int): Int {
    percentFull += increase
    if (percentFull > max)
      percentFull = max
    return percentFull
  }
}

fun main() {
  val cup = Cup2()
  cup.add(50)
  println(cup.percentFull)
  cup.add(70)
```

```
  println(cup.percentFull)
}
/* Výstup:
50
100
*/
```

Členská funkce `add()` se pokusí přidat `nárůst` k `procentůmFull`, ale zajistí, aby nepřekročil 100 %.

Vlastnosti i členské funkce je nutné kvalifikovat zvenčí třídy.

Můžete definovat vlastnosti nejvyšší úrovně:

```
// Properties/TopLevelProperty.kt

hodnota constant = 42

var counter = 0

fun inc() {
  počítadlo++
}
```

Definice `val` nejvyšší úrovně je bezpečná, protože ji nelze změnit. Definování mutovatelné vlastnosti nejvyšší úrovně (`var`) je však považováno za *anti-vzor*. Jak se váš program stává složitějším, je obtížnější správně uvažovat o *sdíleném mutabilním stavu*. Pokud má každý ve vaší kódové bázi přístup k `počítadlu var`, nemůžete zaručit, že se bude správně měnit: zatímco `funkce inc()` zvýší `počítadlo` o jedna, jiná část programu může `počítadlo` snížit o deset, čímž vzniknou nepřehledné chyby. Nejlepší je hlídat proměnlivý stav uvnitř třídy. V kapitole Omezení viditelnosti se dozvíte, jak jej učinit skutečně skrytým.

Tvrdit, že `vars` lze měnit, zatímco `vals` nikoli, je příliš zjednodušující. Jako analogii uvažujme `dům` jako `val` a `pohovku` uvnitř `domu` jako `var`.

`Pohovku` můžete měnit, protože je to `var`. Nemůžete však přeřadit `house`, protože je to `val`:

```
// Properties/ChangingAVal.kt

třída House {
  var sofa: String = ''
}

fun main() {
  val house = House()
  house.sofa = 'Jednoduchá pohovka na spaní: $89,00'
  println(house.sofa)
  house.sofa = 'Nová kožená pohovka: 3 099,00 USD'
  println(house.sofa)
  // Nelze přiřadit hodnotu do nového domu:
  // house = House()
}
/* Výstup:
Jednoduchá pohovka na spaní: $89,00
Nová kožená pohovka: 3 099,00 USD
*/
```

Přestože je `house` `val`, jeho objekt lze modifikovat, protože `sofa` ve `třídě House` je `var`. Definování `house` jako `val` pouze zabraňuje jeho přeřazení do nového objektu.

Pokud vytvoříme vlastnost `val`, nelze ji znovu přiřadit:

```
// Properties/AnUnchangingVar.kt

třída Sofa {
  obálka val: String = 'Potah na křeslo'
}
```

```
fun main() {
  var sofa = Sofa()
  // Není povoleno:
  // sofa.cover = 'Nový kryt'
  // Opětovné přiřazení proměnné:
  sofa = Sofa()
}
```

I když je `sofa` `var`, její objekt nelze změnit, protože `cover` ve `třídě` `Sofa` je `val`. `Sofa` však lze přiřadit novému objektu.

O identifikátorech jako `dům` a `pohovka` jsme mluvili, jako by to byly předměty. Ve skutečnosti jsou to *odkazy na* objekty. Jedním ze způsobů, jak si to uvědomit, je pozorovat, že dva identifikátory mohou odkazovat na stejný objekt:

```
// Properties/References.kt

třída Kitchen {
  var table: String = 'Kulatý stůl'
}

fun main() {
  val kitchen1 = Kitchen()
  val kitchen2 = kitchen1
  println('kitchen1: ${kitchen1.table}')
  println('kitchen2: ${kitchen2.table}')
  kitchen1.table = 'Čtvercový stůl'
  println('kitchen1: ${kitchen1.table}')
  println('kitchen2: ${kitchen2.table}')
}
/* Výstup:
kuchyně1: Kulatý stůl
kuchyně2: Kulatý stůl
```

```
kuchyně1: Čtvercový stůl
kuchyně2: Čtvercový stůl
*/
```

Když `kitchen1` změní `tabulku`, `kitchen2` uvidí změnu. `kitchen1.table` a `kitchen2.table` zobrazí stejný výstup.

Nezapomeňte, že `var` a `val` řídí reference, nikoli objekty. `Var` vám umožňuje převázat referenci na jiný objekt a `val` vám v tom brání.

Mutabilita znamená, že objekt může měnit svůj stav. Ve výše uvedených příkladech definují `třídy House` a `Kitchen` proměnlivé objekty, zatímco `třída Sofa` definuje neměnné objekty.

Konstruktéři

Nový objekt inicializujete předáním informací *konstruktoru*.

Každý objekt je izolovaný svět. Program je kolekcí objektů, takže správná inicializace každého jednotlivého objektu řeší velkou část problému inicializace. Jazyk Kotlin obsahuje mechanismy, které zaručují správnou inicializaci objektů.

Konstruktor je jako speciální členská funkce, která inicializuje nový objekt. Nejjednodušší formou konstruktoru je jednořádková definice třídy:

```
// Constructors/Wombat.kt

třída Wombat

fun main() {
  hodnota wombat = Wombat()
}
```

Voláním funkce `Wombat()` v příkazu `main()` se vytvoří objekt `Wombat`. Pokud přicházíte z jiného objektově orientovaného jazyka, možná byste očekávali, že zde bude použito klíčové slovo `new`, ale `new` by bylo v Kotlinu zbytečné, proto bylo vynecháno.

Informace se konstruktoru předávají pomocí seznamu parametrů, stejně jako funkci. Zde konstruktor `Alien` přijímá jediný argument:

```
// Constructors/Arg.kt

class Alien(name: String) {
  val greeting = 'Chudák $jméno!'
}

fun main() {
  val alien = Alien('Pan Meeseeks')
  println(alien.greeting)
  // alien.name // Error // [1]
}
/* Výstup:
Chudák pan Meeseeks!
*/
```

Vytvoření objektu `Alien` vyžaduje argument (zkuste to bez něj). `name` inicializuje vlastnost `pozdrav v` konstruktoru, ale mimo konstruktor není přístupná - zkuste odkomentovat řádek **[1]**.

Pokud chcete, aby byl parametr konstruktoru přístupný mimo tělo třídy, definujte jej v seznamu parametrů jako `var` nebo `val`:

```
// Constructors/VisibleArgs.kt

class MutableNameAlien(var name: String)

class FixedNameAlien(val name: String)

fun main() {
  hodnota alien1 =
    MutableNameAlien('Reverzní žirafa')
  val alien2 =
```

```
    FixedNameAlien('Krombopolis Michael')

  alien1.name = 'Parazit'
  // To nelze:
  // alien2.name = 'Parasite'
}
```

Tyto definice tříd nemají explicitní těla tříd - těla jsou implicitní.

Pokud je `jméno` definováno jako `var` nebo `val`, stává se vlastností, a je tedy přístupné mimo konstruktor. parametry konstruktoru `val` nelze měnit, zatímco parametry konstruktoru `var` měnit lze.

Vaše třída může mít mnoho parametrů konstruktoru:

```
// Constructors/MultipleArgs.kt

třída AlienSpecies(
  val name: String,
  val oči: Int,
  val ruce: Int,
  hodnota legs: Int
) {
  fun describe() =
    '$jméno s $očima, ' +
      '$ruce ruce a $nohy nohy'
}

fun main() {
  val kevin =
    AlienSpecies('Zigerion', 2, 2, 2)
  val mortyJr =
    AlienSpecies('Gazorpian', 2, 6, 2)
  println(kevin.describe())
  println(mortyJr.describe())
```

```
}
/* Výstup:
Zigerion se 2 očima, 2 rukama a 2 nohama
Gazorpian se 2 očima, 6 rukama a 2 nohama
*/
```

V části Složené konstruktory se dozvíte, že konstruktory mohou obsahovat také složitou inicializační logiku.

Pokud je při očekávání `řetězce` použit objekt, zavolá Kotlin členskou funkci `toString()` objektu. Pokud takovou funkci nenapíšete, stále dostanete výchozí funkci `toString()`:

```
// Constructors/DisplayAlienSpecies.kt

fun main() {
  val krombopulosMichael =
    AlienSpecies('Gromflomite', 2, 2, 2)
  println(krombopulosMichael)
}
/* Ukázka výstupu:
AlienSpecies@4d7e1886
*/
```

Výchozí funkce `toString()` není příliš užitečná - vytváří název třídy a fyzickou adresu objektu (ta se v jednotlivých provedeních programu liší). Můžete si definovat vlastní `toString()`:

```
// Constructors/Scientist.kt

class Scientist(val name: String) {
  override fun toString(): String {
    vrátit 'Scientist('$name')'
  }
}
```

```
fun main() {
  val zeep = Scientist('Zeep Xanflorp')
  println(zeep)
}
/* Výstup:
Vědec('Zeep Xanflorp')
*/
```

`override` je pro nás nové klíčové slovo. Je zde nutné, protože `toString()` již má definici, která vytváří primitivní výsledek. `override` říká Kotlinu, že ano, skutečně chceme nahradit výchozí `toString()` naší vlastní definicí. Explicitnost `override` zpřehledňuje kód a zabraňuje chybám.

Funkce `toString()`, která zobrazuje obsah objektu ve vhodném tvaru, je užitečná při hledání a opravě programátorských chyb. Pro zjednodušení procesu *ladění* poskytují IDE *ladicí programy*, které umožňují sledovat každý krok při provádění programu a nahlížet dovnitř objektů.

Omezení viditelnosti

Pokud kus kódu na několik dní nebo týdnů opustíte a pak se k němu vrátíte, možná zjistíte, že ho lze napsat mnohem lépe.

To je jedna z hlavních motivací pro *refaktoring,* který přepisuje funkční kód tak, aby byl čitelnější, srozumitelnější, a tím i lépe udržovatelný.

Touha měnit a vylepšovat svůj kód je napjatá. Spotřebitelé (*klientští programátoři)* vyžadují, aby některé aspekty vašeho kódu byly stabilní. Vy jej chcete měnit a oni chtějí, aby zůstal stejný.

To je důležité zejména pro knihovny. Uživatelé knihovny nechtějí přepisovat kód pro novou verzi této knihovny. Tvůrce knihovny však musí mít možnost provádět úpravy a vylepšení s jistotou, že klientský kód nebude těmito změnami ovlivněn.

Při návrhu softwaru je proto třeba brát v úvahu především:

Oddělte věci, které se mění, od věcí, které zůstávají stejné.

Kotlin a některé další jazyky poskytují *modifikátory přístupu pro* kontrolu viditelnosti. Tvůrci knihoven pomocí modifikátorů `public`, `private`, `protected` a `internal` rozhodují o tom, co je a co není přístupné klientskému programátorovi. Tato kapitola se zabývá modifikacemi `public` a `private` a stručně se seznámí s modifikací `internal`. `Chráněné` vysvětlíme později v knize.

Před definicí třídy, funkce nebo vlastnosti se objeví modifikátor přístupu, například `private`. Modifikátor přístupu řídí přístup pouze pro danou definici.

`Veřejná` definice je přístupná klientským programátorům, takže změny této definice mají přímý dopad na klientský kód. Pokud modifikátor neuvedete, vaše definice je automaticky `veřejná`, takže `public` je technicky vzato zbytečné. Někdy přesto z důvodu přehlednosti uvedete `public`.

`Soukromá` definice je skrytá a přístupná pouze ostatním členům téže třídy. Změna nebo dokonce odstranění `soukromé` definice nemá přímý dopad na klientské programátory.

`soukromé` třídy, funkce nejvyšší úrovně a vlastnosti nejvyšší úrovně jsou přístupné pouze uvnitř tohoto souboru:

```
// Visibility/RecordAnimals.kt

private var index = 0 // [1]

private class Animal(val name: String) // [2]

private fun recordAnimal( // [3]
  zvíře: Zvíře
) {
  println('Zvíře #$index: ${zvíře.jméno}')
  index++
```

```
}

fun recordAnimals() {
  recordAnimal(Zvíře('Tiger'))
  recordAnimal(Zvíře('Antelope'))
}

fun recordAnimalsCount() {
  println('$index zvířata jsou zde!')
}
```

K `soukromým` vlastnostem nejvyšší úrovně (**[1]**), třídám (**[2]**) a funkcím (**[3]**) můžete přistupovat z jiných funkcí a tříd v rámci `souboru RecordAnimals.kt`. Kotlin vám brání v přístupu k `soukromému` prvku nejvyšší úrovně z jiného souboru tím, že vám v něm sdělí, že je `soukromý`:

```
// Visibility/ObserveAnimals.kt

fun main() {
  // Nelze přistupovat k soukromým členům
  // deklarováno v jiném souboru.
  // Třída je soukromá:
  // val rabbit = Animal('Rabbit')
  // Funkce je soukromá:
  // recordAnimal(králík)
  // Vlastnost je soukromá:
  // index++

  recordAnimals()
  recordAnimalsCount()
}
/* Výstup:
Zvíře #0: Tygr
Zvíře č. 1: antilopa
```

```
Jsou tu 2 zvířata!
*/
```

Soukromí se nejčastěji používá pro členy třídy:

```
// Visibility/Cookie.kt

třída Cookie(
  private var isReady: [1]
) {
  private fun crumble() = // [2]
    println('crumble')

  public fun bite() = // [3]
    println('bite')

  fun eat() { // [4]
    isReady = true // [5]
    rozpadnout()
    kousnutí()
  }
}

fun main() {
  hodnota x = Cookie(false)
  x.bite()
  // Nelze přistupovat k soukromým členům:
  // x.isReady
  // x.crumble()
  x.eat()
}
/* Výstup:
kousnutí
```

```
drobit
kousnutí
*/
```

- **[1]** `Soukromá` vlastnost, která není přístupná mimo třídu, která ji obsahuje.
- **[2]** `Soukromá` členská funkce.
- **[3]** `Veřejná` členská funkce, přístupná komukoli.
- **[4]** Bez modifikátoru přístupu znamená `veřejný`.
- **[5]** **K** `soukromým` členům mají přístup pouze členové stejné třídy.

Klíčové slovo `private` znamená, že k danému členu nemá přístup nikdo jiný než ostatní členové dané třídy. Ostatní třídy k `privátním` členům přistupovat nemohou, takže jako byste třídu izolovali i proti sobě a svým spolupracovníkům. Pomocí `private` můžete daný člen libovolně měnit, aniž byste se museli starat o to, zda to ovlivní jinou třídu ve stejném balíku. Jako návrhář knihovny budete obvykle udržovat věci co nejvíce `privátní` a klientským programátorům vystavíte pouze funkce a třídy.

Jakoukoli členskou funkci, která je *pomocnou funkcí* třídy, lze učinit `soukromou`, abyste zajistili, že ji omylem nepoužijete jinde v balíku, a tím si znemožnili tuto funkci změnit nebo odstranit.

Totéž platí pro `soukromou` vlastnost uvnitř třídy. Pokud není nutné odhalit základní implementaci (což je méně pravděpodobné, než si myslíte), nastavte vlastnosti jako `soukromé`. Nicméně to, že je odkaz na objekt uvnitř třídy `soukromý`, neznamená, že nějaký jiný objekt nemůže mít `veřejný` odkaz na stejný objekt:

```
// Visibility/MultipleRef.kt

třída Counter(var start: Int) {
  fun increment() {
    start += 1
  }
  override fun toString() = start.toString()
}
```

```
třída CounterHolder(counter: Counter) {
  private val ctr = counter
  override fun toString() =
    'CounterHolder: ' + ctr
}

fun main() {
  val c = Counter(11) // [1]
  val ch = CounterHolder(c) // [2]
  println(ch)
  c.increment() // [3]
  println(ch)
  val ch2 = CounterHolder(Counter(9)) // [4]
  println(ch2)
}
/* Výstup:
CounterHolder: 11
CounterHolder: 12
CounterHolder: 9
*/
```

- **[1]** `c` je nyní definováno v oboru *kolem* vytvoření objektu `CounterHolder` na následujícím řádku.
- **[2]** Předání `c` jako argumentu konstruktoru `CounterHolder` znamená, že nový `CounterHolder` nyní odkazuje na stejný objekt `Counter`, na který odkazuje `c`.
- **[3]** **S** `čítačem`, který je uvnitř `ch` údajně `soukromý`, lze přesto manipulovat pomocí `c`.
- **[4]** `Counter(9)` nemá žádné jiné reference než v rámci `CounterHolderu`, takže k němu nemůže přistupovat ani ho měnit nic jiného než `ch2`.

Udržování více odkazů na jeden objekt se nazývá *aliasing* a může vést k překvapivému chování.

Moduly

Na rozdíl od malých příkladů v této knize jsou skutečné programy často velké. Může být užitečné rozdělit takové programy do jednoho nebo více *modulů*. Modul je logicky nezávislá část kódové základny. Způsob rozdělení projektu do modulů závisí na systému sestavování (například Gradle nebo Maven) a přesahuje rámec této knihy.

`Interní` definice je přístupná pouze v rámci modulu, kde je definována. `interní je` něco mezi `privátní` a `veřejnou - používá se,` pokud je `privátní` příliš omezující, ale nechcete, aby byl prvek součástí `veřejného` API. V příkladech a cvičeních v knize `internal` nepoužíváme.

Moduly jsou konceptem vyšší úrovně. Následující kapitola představuje *balíčky,* které umožňují jemnější strukturování. Knihovna je často jeden modul skládající se z více balíčků, takže `vnitřní` prvky jsou dostupné v rámci knihovny, ale nejsou přístupné konzumentům této knihovny.

Balíčky

Základním principem programování je zkratka DRY: *Don't Repeat Yourself (neopakuj se)*.

Více stejných částí kódu vyžaduje údržbu, kdykoli provedete opravy nebo vylepšení. Duplikace kódu tedy není jen práce navíc - každá duplikace vytváří příležitosti k chybám.

Klíčové slovo `import` znovu použije kód z jiných souborů. Jedním ze způsobů použití `importování` je zadání názvu třídy, funkce nebo vlastnosti:

```
import packagename.ClassName
import packagename.functionName
import packagename.propertyName
```

Balíček je přidružená kolekce kódu. Každý balíček je obvykle určen k řešení určitého problému a často obsahuje více funkcí a tříd. Můžeme například importovat matematické konstanty a funkce z knihovny `kotlin.math`:

```
// Packages/ImportClass.kt
import kotlin.math.PI
import kotlin.math.cos // Cosine

fun main() {
  println(PI)
  println(cos(PI))
  println(cos(2 * PI))
}
/* Výstup:
3.141592653589793
-1.0
1.0
*/
```

Někdy chcete použít více knihoven třetích stran obsahujících třídy nebo funkce se stejným názvem. Klíčové slovo `as` vám umožní měnit názvy při importu:

```
// Packages/ImportNameChange.kt
import kotlin.math.PI as circleRatio
import kotlin.math.cos as cosine

fun main() {
  println(circleRatio)
  println(cosine(circleRatio))
  println(cosine(2 * circleRatio))
}
/* Výstup:
3.141592653589793
```

```
-1.0
1.0
*/
```

což je užitečné, pokud je název knihovny špatně zvolený nebo příliš dlouhý.

Import můžete plně kvalifikovat v těle kódu. V následujícím příkladu může být kód kvůli explicitním názvům balíčků méně čitelný, ale původ jednotlivých prvků je naprosto jasný:

```
// Packages/FullyQualify.kt

fun main() {
  println(kotlin.math.PI)
  println(kotlin.math.cos(kotlin.math.PI))
  println(kotlin.math.cos(2 * kotlin.math.PI))
}
/* Výstup:
3.141592653589793
-1.0
1.0
*/
```

Chcete-li importovat vše z balíčku, použijte hvězdičku:

```
// Packages/ImportEverything.kt
import kotlin.math.*

fun main() {
  println(E)
  println(E.roundToInt())
  println(E.toInt())
}
/* Výstup:
2.718281828459045
```

```
3
2
*/
```

Balíček `kotlin.math` obsahuje praktickou funkci `roundToInt()`, která zaokrouhlí hodnotu `Double na` nejbližší celé číslo, na rozdíl od `funkce toInt(), která` jednoduše zkrátí cokoli za desetinnou čárkou.

Chcete-li svůj kód použít opakovaně, vytvořte balíček pomocí klíčového slova `package.` Příkaz `package` musí být prvním nekomentovaným příkazem v souboru. za příkazem `package` následuje název vašeho balíčku, který se podle konvence píše malými písmeny:

```
// Packages/PythagoreanTheorem.kt
balíček pythagorean
import kotlin.math.sqrt

třída RightTriangle(
  val a: Double,
  val b: Double
) {
  fun hypotenuse() = sqrt(a * a + b * b)
  fun area() = a * b / 2
}
```

Na rozdíl od Javy, která vyžaduje, aby se název souboru shodoval s názvem třídy, můžete soubor se zdrojovým kódem pojmenovat jakkoli.

Jazyk Kotlin umožňuje zvolit si libovolný název balíčku, ale za dobrý styl se považuje, aby byl název balíčku shodný s názvem adresáře, kde jsou soubory balíčku umístěny (v příkladech v této knize tomu tak vždy nebude).

Prvky v balíčku `pythagorean` jsou nyní dostupné pomocí `importu`:

```
// Packages/ImportPythagorean.kt
import pythagorean.RightTriangle
```

```
fun main() {
  hodnota rt = Pravoúhlý trojúhelník(3.0, 4.0)
  println(rt.hypotenuse())
  println(rt.area())
}
/* Výstup:
5.0
6.0
*/
```

Ve zbytku této knihy budeme používat příkazy `package pro` všechny soubory, které definují funkce, třídy atd. mimo `main()`, `abychom` zabránili kolizi názvů s jinými soubory v knize, ale obvykle nebudeme vkládat příkaz `package do` souboru, který obsahuje *pouze* `main()`.

Vytvořeno v rámci projektu: **DANTE**, reg. č. NPO_UPCE_MSMT-16591/2022

Financováno Evropskou unií

NextGenerationEU

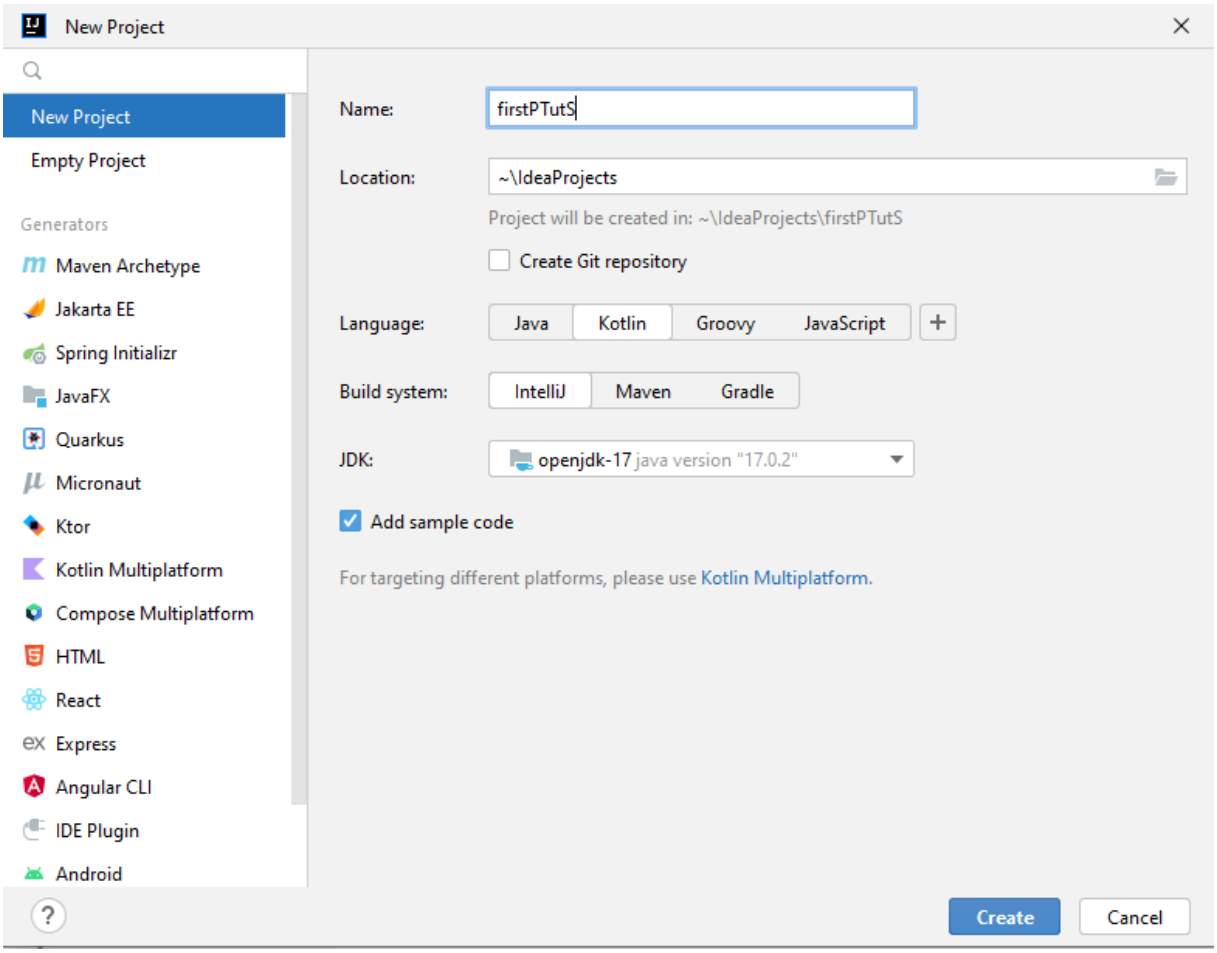
Národní plán obnovy

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

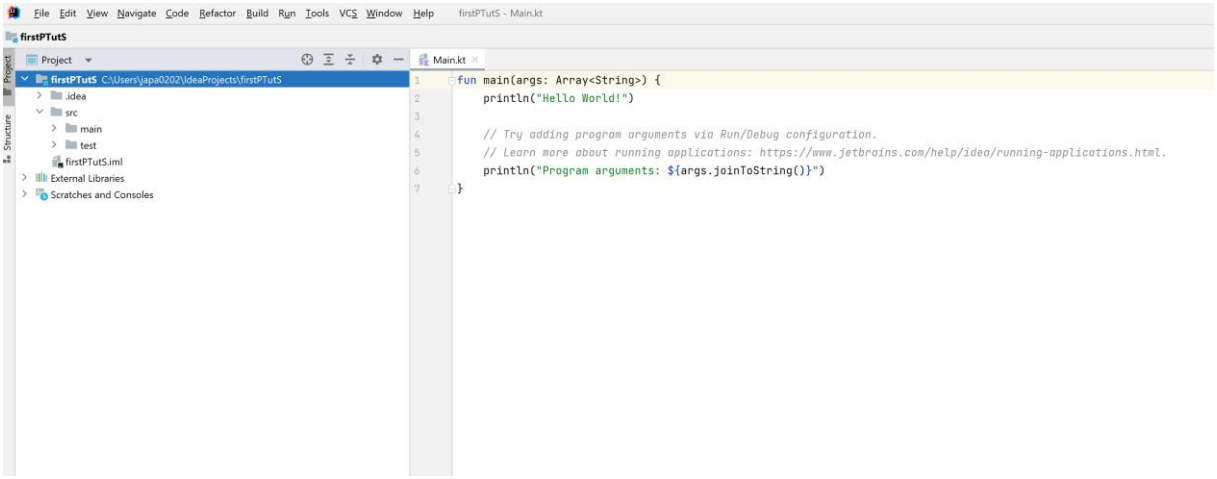
První program!

Pro psaní kódu v programovacím jazyku Kotin existuje celá řada textových editorů. Jedním z nejpoužívanějším je IntelliJ Idea od firmy JetBrains. Druhou možností je psát kód přímo v Android Studiu. V úvodních kapitolách si ukážeme práci v aplikaci IntelliJ Idea a později přejdeme do Android Studia.

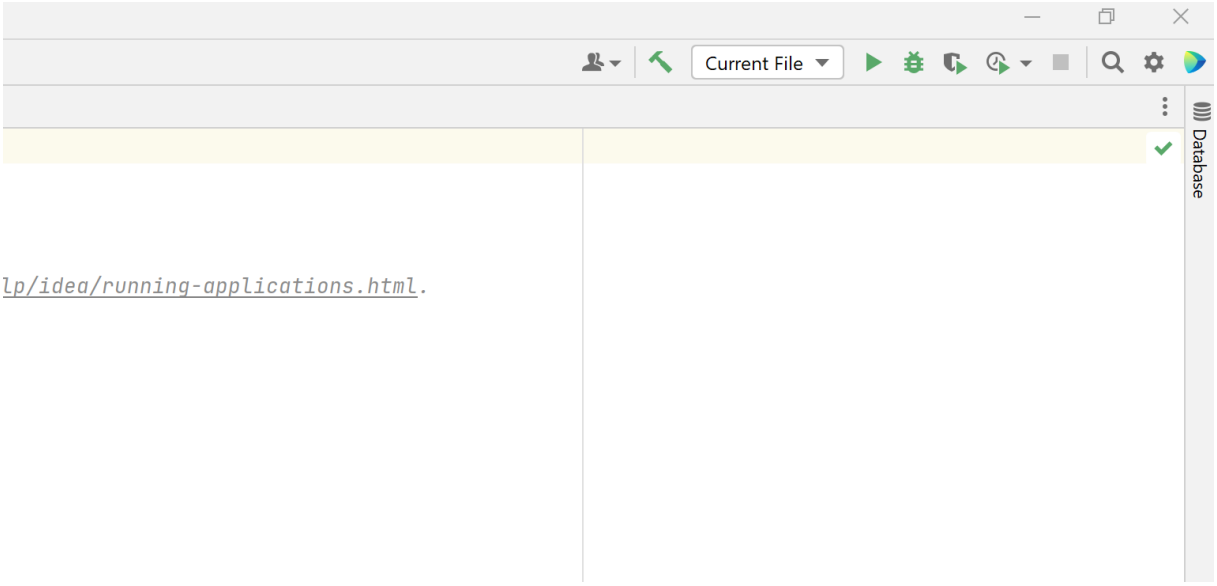
Na začátek si otevřeme program a vytvoříme si nový projekt: File – New Project:



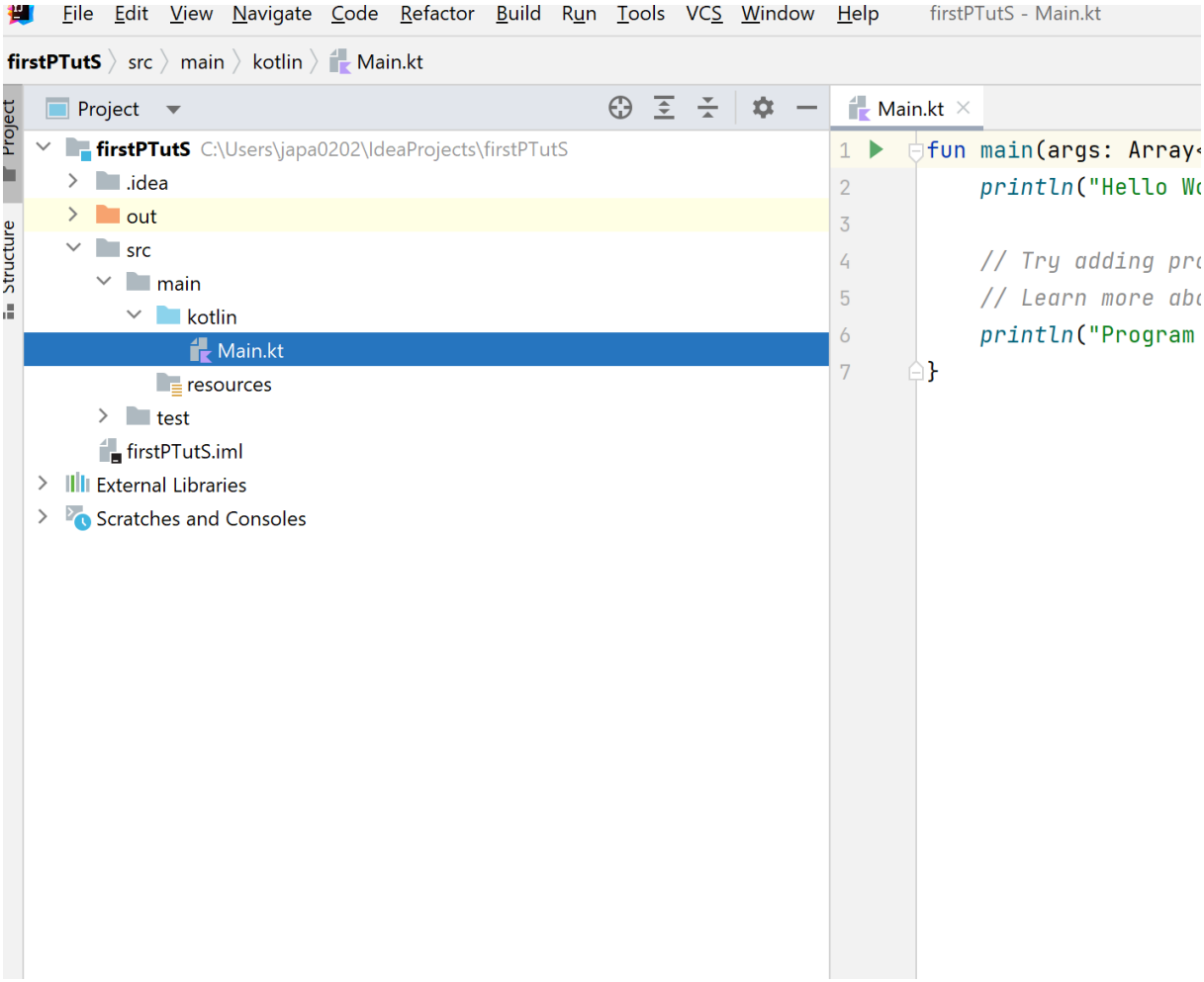
Projekt si nějak pojmenujeme – zadáme, že se jedná o console application, tedy o konzolovou aplikaci a jako programovací jazyk zadáme Kotlin. Build systém necháme Gradle Kotlin – ze začátku jiný používat nebudeme. Pokud máte zkušenosti s jiným, můžete si vybrat ten. JDK si vyberte podle aktuálně instalované JAVA, které máte na osobním počítači.



Nyní můžeme zadávat kód a nechat si program spustit, program se spouští buď v menu Run – Run (Alt+Shift+F10) nebo v pravém horním rohu zelenou šipkou.



Strukturu programu naleznete zde:



Tento program vyvíjíme v několika krocích, abyste porozuměli jeho částem.

```
fun main() {
   // Zde naprogramujte kód ...
}
```

Příklad začíná *komentářem*, což je osvětlující text, který Kotlin ignoruje. `//` (dvě lomítka vpřed) začíná komentář, který vede na konec aktuálního řádku:

```
// Jednořádkový komentář
```

Kotlin ignoruje znak `//` a vše za ním až do konce řádku. Na následujícím řádku opět zpozorní.

klíčová slova jsou v jazyce vyhrazena a mají zvláštní význam. Klíčové slovo `fun` je zkratka pro *funkci*. Funkce je soubor kódu, který lze spustit pomocí

názvu této funkce (funkcím se v celé knize budeme věnovat hodně času). Název funkce následuje za klíčovým slovem `fun`, takže v tomto případě je to `main()` (v próze následuje za názvem funkce závorka).

`main()` je vlastně speciální název pro funkci; označuje "vstupní bod" programu Kotlin. Program v jazyce Kotlin může mít mnoho funkcí s mnoha různými názvy, ale `main()` je ta, která se automaticky zavolá při spuštění programu.

Seznam parametrů následuje za názvem funkce a je uzavřen v závorkách. Zde do `funkce main()` nic nepředáváme, takže seznam parametrů je prázdný.

Tělo funkce se zobrazí za seznamem parametrů. Začíná otevírací závorkou (`{`) a končí uzavírací závorkou (`}`). Tělo funkce obsahuje *příkazy* a *výrazy*. Příkaz vytváří efekt a výraz dává výsledek.

`Main.kt` obsahuje v těle základní příkazy pro vypsání slov „Hello World!“, komentáře a „Program arguments“.

Řádek, který zobrazí pozdrav, začíná příkazem `println()`. Stejně jako `main()` je `println()` funkce. Tento řádek *volá* funkci, která provede své tělo. Uvádí se název funkce, za nímž následují závorky obsahující jeden nebo více parametrů. V této knize při odkazování na funkci v próze přidáváme závorky za název jako připomínku, že se jedná o funkci. Zde říkáme `println()`.

`println()` přijímá jediný parametr, kterým je `řetězec String`. `Řetězec` se definuje tak, že se znaky vloží do uvozovek.

`println()` po zobrazení svého parametru přesune kurzor na nový řádek, takže následný výstup se zobrazí na dalším řádku. Místo toho můžete použít `funkci print()`, která ponechá kurzor na stejném řádku.

Na rozdíl od některých jazyků nepotřebujete v Kotlinu na konci výrazu středník. Je nutný pouze v případě, že na jeden řádek umístíte více než jeden výraz (to se nedoporučuje).

U některých příkladů v knize uvádíme výstup na konci výpisu, uvnitř *víceřádkového komentáře*. Víceřádkový komentář začíná znakem `/*` (lomítko vpřed následované hvězdičkou) a pokračuje - včetně zlomů řádků (kterým říkáme *nové řádky)* - až do konce komentáře znakem `*/` (hvězdička následovaná lomítkem vpřed):

```
/* Víceřádkový komentář
Nezajímá se o to
o nových řádcích */
```

Je možné přidat kód na stejný řádek *za* uzavírací `*/` komentáře, ale je to matoucí, takže to lidé obvykle nedělají.

Komentáře přidávají informace, které nejsou zřejmé ze čtení kódu. Pokud komentáře pouze opakují to, co je uvedeno v kódu, začnou být otravné a lidé je začnou ignorovat. Při změnách kódu programátoři často zapomínají komentáře aktualizovat, proto je dobré používat komentáře uvážlivě, především pro zvýraznění záludných aspektů kódu.

`var` & `val`

Pokud identifikátor obsahuje data, je třeba rozhodnout, zda jej lze znovu přiřadit.

Vytváříte *identifikátory, které* odkazují na prvky v programu. Nejzákladnějším rozhodnutím pro datový identifikátor je, zda může měnit svůj obsah během provádění programu, nebo zda může být přiřazen pouze jednou. To se řídí dvěma klíčovými slovy:

- `var`, což je zkratka pro *proměnnou*, což znamená, že její obsah můžete přiřadit.
- `val`, zkratka pro *value*, což znamená, že ji můžete pouze inicializovat; nemůžete ji znovu přiřadit.

Definujete `var` takto:

```
var identifier = inicializace
```

Za klíčovým slovem `var` následuje identifikátor, rovnítko a poté inicializační hodnota. Identifikátor začíná písmenem nebo podtržítkem, za ním následují písmena, číslice a podtržítka. Rozlišují se velká a malá písmena (takže `thisvalue` a `thisValue` se liší).

Zde je několik definic `var`:

```
fun main() {
  var whole = 11 // [1]
  var fractional = 1.4 // [2]
  var words = 'Twas Brillig' // [3]
  println(celá)
  println(zlomek)
  println(slova)
}
/* Výstup:
11
1.4
Twas Brillig
*/
```

V této knize označujeme řádky komentovanými čísly v hranatých závorkách, abychom na ně mohli v textu takto odkazovat:

- **[1]** Vytvořte `var s` názvem `whole` a uložte do něj `11`.
- **[2]** Do `var fractional` uložte "zlomkové číslo" `1,4`.
- **[3]** Do `var words` uložte nějaký text (`řetězec`).

Všimněte si, že `funkce println()` může jako argument přijmout libovolnou hodnotu.

Jak už název *proměnná* napovídá, `var` se může lišit. To znamená, že data uložená ve `var` můžete měnit. Říkáme, že `var` je *proměnná*:

```
fun main() {
```

```
  var sum = 1
  sum = sum + 2
  sum += 3
  println(sum)
}
/* Výstup:
6
*/
```

Přiřazení `sum = sum + 2` vezme aktuální hodnotu `součtu`, přičte dvě a výsledek přiřadí zpět do `součtu`.

Přiřazení `sum += 3` znamená totéž co `sum = sum + 3`. Operátor `+=` vezme předchozí hodnotu uloženou v `součtu`, zvýší ji o `3` a tento nový výsledek přiřadí zpět do `součtu`.

Změna hodnoty uložené ve `var` je užitečný způsob, jak vyjádřit změny. Když se však složitost programu zvyšuje, je kód přehlednější, bezpečnější a srozumitelnější, pokud se hodnoty reprezentované identifikátory nemohou měnit - to znamená, že je nelze znovu přiřadit. Neměnný identifikátor zadáváme pomocí klíčového slova `val` místo `var`. `Val` lze přiřadit pouze jednou, a to při jeho vytvoření:

```
val identifier = inicializace
```

Klíčové slovo `val` pochází z *value* a označuje něco, co se nemůže změnit - je *neměnné*. Kdykoli je to možné, volte místo slova `var slovo val`. Příklad `Vars.kt` na začátku tohoto atomu lze přepsat pomocí vals:

```
fun main() {
  hodnota whole = 11
  // celá = 15 // Chyba // [1]
  val fractional = 1.4
  val words = 'Twas Brillig'
  println(celá)
  println(zlomek)
```

```
  println(slova)
}
/* Výstup:
11
1.4
Twas Brillig
*/
```

- **[1]** Jakmile jednou inicializujete `hodnotu`, nemůžete ji znovu přiřadit. Pokud se pokusíme o nové přiřazení `celku` na jiné číslo, Kotlin si stěžuje a říká "Val nelze znovu přiřadit".

Volba popisných názvů pro identifikátory usnadňuje pochopení kódu a často snižuje potřebu komentářů. V souboru `Vals.kt` netušíte, co představuje `celek`. Pokud váš program ukládá číslo 11, které reprezentuje denní dobu, kdy si dáváte kávu, bude pro ostatní zřejmější, když ho pojmenujete `coffeetime`, a snáze čitelné, když bude `coffeeTime` (podle stylu Kotlinu děláme první písmeno malé).

- -

vars jsou užitečné, pokud se data musí měnit za běhu programu. To zní jako běžný požadavek, ale v praxi se ukazuje, že se mu lze vyhnout. Obecně lze říci, že vaše programy se snadněji rozšiřují a udržují, pokud používáte vals. Ve výjimečných případech je však řešení problému pouze pomocí vals příliš složité. Z tohoto důvodu vám Kotlin poskytuje flexibilitu vars. Jakmile však strávíte s vals více času, zjistíte, že vars téměř nikdy nepotřebujete a že vaše programy jsou bezpečnější a spolehlivější bez nich.

Datové typy

Data mohou mít různé *typy*.

Při řešení matematického problému napíšete výraz:

```
5.9 + 6
```

Víte, že součtem těchto čísel získáte další číslo. Kotlin to ví také. Víte, že jedno je zlomkové číslo (`5,9`), které Kotlin nazývá `Double`, a druhé je celé číslo (`6`), které Kotlin nazývá `Int`. Víte, že výsledek je zlomkové číslo.

Typ (nazývaný také *datový typ*) říká jazyku Kotlin, jak chcete daná data používat. Typ poskytuje množinu hodnot, ze kterých může výraz přebírat své hodnoty. Typ definuje operace, které lze s daty provádět, význam dat a způsob, jakým lze hodnoty daného typu ukládat.

Kotlin používá typy k ověření správnosti výrazů. Ve výše uvedeném výrazu vytvoří Kotlin novou hodnotu typu `Double`, která bude obsahovat výsledek.

Kotlin se snaží přizpůsobit tomu, co potřebujete. Pokud jej požádáte o něco, co porušuje typová pravidla, zobrazí chybovou zprávu. Zkuste například přidat `řetězec String` a číslo:

```
// DataTypes/StringPlusNumber.kt


fun main() {
  println('Sally' + 5.9)
}
/* Výstup:
Sally5.9
*/
```

Typy říkají jazyku Kotlin, jak je správně používat. V tomto případě typová pravidla říkají Kotlinu, jak přidat číslo k `řetězci`: přičtením obou hodnot a vytvořením `řetězce`, který bude obsahovat výsledek.

Nyní zkuste vynásobit `řetězec` a `dvojčíslí` tak, že změníte `+` v souboru `StringPlusNumber.kt` na `*`:

```
"Sally" * 5,9
```

Kombinování typů tímto způsobem nedává Kotlinu smysl, a proto vám vyhodí chybu.

Do `var` & `val` jsme uložili několik typů. Kotlin za nás typy určil na základě toho, jak jsme je používali. Tomu se říká *typová inference*.

Můžeme být stručnější a zadat typ:

```
val identifier: Typ = inicializace
```

Začínáte klíčovým slovem `val` nebo `var`, následuje identifikátor, dvojtečka, typ, `=` a inicializační hodnota. Takže místo toho, abyste řekli:

```
hodnota n = 1
var p = 1.2
```

Můžete říci:

```
val n: Int = 1
var p: Double = 1.2
```

Kotlinu jsme řekli, že `n` je `Int` a `p` je `Double`, místo abychom ho nechali typ odvodit.

Zde jsou některé základní typy jazyka Kotlin:

```
// DataTypes/Types.kt

fun main() {
  val celek: Int = 11 // [1]
  val fractional: = 1.4 // [2]
  val trueOrFalse: [3]: Boolean = true // [3]
  val slova: String = 'A value' // [4]
  val character: = 'z' // [5]
  val lines: String = '''Trojité uvozovky let
máte mnoho řádků
ve vašem řetězci''' // [6]
  println(celá)
  println(zlomek)
  println(trueOrFalse)
  println(slova)
```

```
  println(znak)
  println(řádky)
}
/* Výstup:
11
1.4
Pravda
Hodnota
z
Trojité uvozovky nechat
máte mnoho řádků
ve vašem řetězci
*/
```

- **[1]** Datový typ `Int` je *celé číslo*, což znamená, že obsahuje pouze celá čísla.
- **[2] Pro** uložení zlomkových čísel použijte `Double`.
- **[3]** Datový typ `Boolean` obsahuje pouze dvě speciální hodnoty `true` a `false`.
- **[4]** `Řetězec` je posloupnost znaků. Hodnotu přiřadíte pomocí `řetězce s` dvojitými uvozovkami.
- **[5]** `Znak Char` obsahuje jeden znak.
- **[6]** Pokud máte mnoho řádků a/nebo speciálních znaků, obklopte je trojitými dvojitými uvozovkami (jedná se o *`řetězec s` trojitými uvozovkami)*.

Kotlin používá k určení významu smíšených typů typovou inferenci. Například při míchání Ints a Doubles během sčítání Kotlin rozhoduje o typu výsledné hodnoty:

```
// DataTypes/Inference.kt

fun main() {
  hodnota n = 1 + 1.2
  println(n)
```

```
}
/* Výstup:
2.2
*/
```

Při přičítání `Int` k `Double` pomocí typové inference Kotlin určí, že výsledek `n` je `Double`, a zajistí, že se bude řídit všemi pravidly pro Double.

Odvozování typů v jazyce Kotlin je součástí jeho strategie, která spočívá v tom, že dělá práci za programátora. Pokud vynecháte deklaraci typu, Kotlin ji obvykle dokáže odvodit.

Funkce

Funkce je jako malý program, který má své vlastní jméno a může být spuštěn (*vyvolán)* zavoláním tohoto jména z jiné funkce.

Funkce spojuje skupinu činností a představuje nejzákladnější způsob organizace programů a opakovaného použití kódu.

Do funkce předáte informace a funkce je použije k výpočtu a vytvoření výsledku. Základní tvar funkce je:

```
fun functionName(p1: Type1, p2: Type2, ...):
ReturnType {
  řádky kódu
  vrátit výsledek
}
```

`p1` a `p2` jsou *parametry*: informace, které předáváte do funkce. Každý parametr má identifikační jméno (`p1`, `p2`), za kterým následuje dvojtečka a typ daného parametru. Za uzavírací závorkou seznamu parametrů následuje dvojtečka a typ výsledku, který funkce produkuje. Řádky kódu v *těle funkce* jsou uzavřeny v kroucených závorkách. Výraz následující za klíčovým slovem `return` je výsledek, který funkce po dokončení vytvoří.

Parametr je způsob, jak definovat, co se předává do funkce - je to zástupný symbol. Argument je skutečná hodnota, kterou předáváte do funkce.

Kombinace názvu, parametrů a návratového typu se nazývá *signatura funkce*.

Zde je jednoduchá funkce s názvem `multiplyByTwo()`:

```
// Funkce/MultiplyByTwo.kt

fun multiplyByTwo(x: Int): [1]
  println('Uvnitř multiplyByTwo') // [2]
  vrátit x * 2
}

fun main() {
  val r = multiplyByTwo(5) // [3]
  println(r)
}
/* Výstup:
Uvnitř multiplyByTwo
10
*/
```

- **[1]** Všimněte si klíčového slova `fun`, názvu funkce a seznamu parametrů, který se skládá z jediného parametru. Tato funkce přijímá parametr `Int` a vrací `Int`.
- **[2]** Tyto dva řádky jsou tělem funkce. Poslední řádek vrací hodnotu jejího výpočtu `x * 2` jako výsledek funkce.
- **[3]** Tento řádek *volá* funkci s příslušným argumentem a výsledek zachytí do `val r`. Volání funkce kopíruje tvar její deklarace: jméno funkce, za nímž následují argumenty v závorkách.

Kód funkce se provede zavoláním funkce, přičemž název funkce `multiplyByTwo()` se použije jako zkratka pro tento kód. Proto jsou funkce nejzákladnější formou zjednodušení a opakovaného použití kódu v programování. Funkci si také můžete představit jako výraz se zastupitelnými hodnotami (parametry).

`println()` je také volání funkce - náhodou ji poskytuje Kotlin. Funkce definované jazykem Kotlin označujeme jako *knihovní funkce*.

Pokud funkce neposkytuje smysluplný výsledek, je její návratový typ `Unit`. Pokud chcete, můžete typ `Unit` zadat explicitně, ale Kotlin jej umožňuje vynechat:

```
// Funkce/SayHello.kt

fun sayHello() {
  println('Hallo!')
}

fun sayGoodbye(): Unit {
  println('Auf Wiedersehen!')
}

fun main() {
  sayHello()
  sayGoodbye()
}
/* Výstup:
Haló!
Auf Wiedersehen!
*/
```

`Funkce sayHello()` i `sayGoodbye()` vracejí `jednotku`, ale funkce `sayHello()` vynechává explicitní deklaraci. Funkce `main()` rovněž vrací `jednotku`.

Pokud je funkce tvořena pouze jedním výrazem, můžete použít zkrácenou syntaxi se znaménkem rovnosti, za kterým následuje výraz:

```
fun functionName(arg1: Type1, arg2: Type2, ...): =
výraz
```

Tělo funkce obklopené kudrnatými závorkami se nazývá *tělo bloku*. Tělo funkce se syntaxí rovnítka se nazývá *tělo výrazu*.

Zde `funkce multiplyByThree()` používá tělo výrazu:

```
// Funkce/MultiplyByThree.kt

fun multiplyByThree(x: Int): Int = x * 3

fun main() {
  println(multiplyByThree(5))
}
/* Výstup:
15
*/
```

Jedná se o zkrácenou verzi příkazu `return x * 3` uvnitř těla bloku.

Kotlin odvozuje návratový typ funkce, která má tělo výrazu:

```
// Funkce/MultiplyByFour.kt

fun multiplyByFour(x: Int) = x * 4

fun main() {
  val výsledek: Int = multiplyByFour(5)
  println(result)
}
/* Výstup:
20
*/
```

Kotlin odvodí, že `funkce multiplyByFour()` vrací `hodnotu Int`.

Kotlin umí odvozovat návratové typy *pouze* pro těla výrazů. Pokud má funkce tělo bloku a vynecháte jeho typ, funkce vrátí `jednotku`.

- -

Při psaní funkcí volte popisné názvy. Kód se tak lépe čte a často se sníží potřeba komentářů. U názvů funkcí v této knize nemůžeme být vždy tak popisní, jak bychom si přáli, protože jsme omezeni šířkou řádku.

`if` Výrazy

Výraz `if` umožňuje volbu.

Klíčové slovo `if` testuje výraz, zda je `pravdivý` nebo `nepravdivý`, a na základě výsledku provede akci. Výraz typu pravda nebo nepravda se nazývá *booleovský* výraz podle matematika George Boolea, který logiku těchto výrazů vymyslel. Zde je příklad s použitím symbolů `>` (větší než) a `<` (menší než):

```
// IfExpressions/If1.kt

fun main() {
  if (1 > 0)
    println('Je to pravda!')
  if (10 < 11) {
    println('10 < 11')
    println('deset je méně než jedenáct')
  }
}
/* Výstup:
Je to pravda!
10 < 11
deset je méně než jedenáct
*/
```

Výraz uvnitř závorek za `if` se musí vyhodnotit jako `true` nebo `false`. Pokud je `true`, provede se následující výraz. Chcete-li provést více řádků, umístěte je do složených závorek.

Na jednom místě můžeme vytvořit logický výraz a na jiném místě jej použít:

```
// IfExpressions/If2.kt

fun main() {
  val x: x: boolean = 1 >= 1
  if (x)
    println('Je to pravda!')
}
/* Výstup:
Je to pravda!
*/
```

Protože `x` je `logická hodnota`, může ji `if` testovat přímo příkazem `if(x)`.

Logický operátor `>=` vrací hodnotu `true`, pokud je výraz na levé straně operátoru *větší nebo* roven výrazu na pravé straně. Podobně operátor `<=` vrací hodnotu `true`, pokud je výraz na levé straně *menší nebo roven* výrazu na pravé straně.

Klíčové slovo `else` umožňuje zpracovávat cesty `true` i `false`:

```
// IfExpressions/If3.kt

fun main() {
  val n: Int = -11
  if (n > 0)
    println('Je to pozitivní')
  jinak
    println('Je záporná nebo nulová')
}
/* Výstup:
Je záporná nebo nulová
*/
```

Klíčové slovo `else` se používá pouze ve spojení s `if`. Nejste omezeni na jedinou kontrolu - kombinací `else` a `if` můžete testovat více kombinací:

```
// IfExpressions/If4.kt

fun main() {
  val n: Int = -11
  if (n > 0)
    println('Je to pozitivní')
  else if (n == 0)
    println('Je to nula')
  jinak
    println('Je to záporné')
}
/* Výstup:
Je negativní
*/
```

Zde používáme `==` ke kontrole rovnosti dvou čísel. `!=` testuje nerovnost.

Typickým vzorem je začít klauzulí `if`, následovat tolika klauzulemi `else if`, kolik potřebujete, a končit závěrečnou `else` pro vše, co neodpovídá všem předchozím testům. Když výraz `if` dosáhne určité velikosti a složitosti, pravděpodobně místo něj použijete výraz `when`. `when` je popsáno později v knize, v části Výrazy `when`.

Operátor "not" `!` testuje opak logického výrazu:

```
// IfExpressions/If5.kt

fun main() {
  val y: Boolean = false
  if (!y)
    println('!y je true')
}
```

```
/* Výstup:
!y je pravda
*/
```

Chcete-li verbalizovat `if(!y)`, řekněte "if not y".

Celé `if` je výraz, takže může vést k výsledku:

```
// IfExpressions/If6.kt

fun main() {
  hodnota num = 10
  val result = if (num > 100) 4 else 42
  println(result)
}
/* Výstup:
42
*/
```

Zde ukládáme hodnotu vytvořenou celým výrazem `if` do mezilehlého identifikátoru nazvaného `result`. Pokud je podmínka splněna, první větev vytvoří `výsledek`. Pokud ne, stane se `výsledkem` hodnota `else`.

Procvičíme si vytváření funkcí. Zde je jedna, která přijímá logický parametr:

```
// IfExpressions/TrueOrFalse.kt

fun trueOrFalse(exp: Boolean): String {
  if (exp)
    vrátit 'Je to pravda!'          // [1]
  return 'It's false' // [2]
}

fun main() {
  hodnota b = 1
  println(trueOrFalse(b < 3))
```

```
  println(trueOrFalse(b >= 3))
}
/* Výstup:
Je to pravda!
Je to falešné
*/
```

Funkci `trueOrFalse()` se předává logický parametr `exp`. Pokud je argument předán jako výraz, například `b < 3`, je tento výraz nejprve vyhodnocen a výsledek je předán funkci. `trueOrFalse()` otestuje `exp`, a pokud je výsledek `pravdivý`, provede se řádek **[1]**, jinak se provede řádek **[2]**.

- **[1]** `return` říká: "Opusťte funkci a vytvořte tuto hodnotu jako výsledek funkce." Všimněte si, že `return` se může objevit kdekoli ve funkci a nemusí být na jejím konci.

Namísto použití `return` jako v předchozím příkladu můžete použít klíčové slovo `else`, které vytvoří výsledek jako výraz:

```
// IfExpressions/OneOrTheOther.kt

fun oneOrTheOther(exp: Boolean): String =
  if (exp)
    "To je pravda! // Není nutné používat 'return'
  jinak
    'False'

fun main() {
  hodnota x = 1
  println(oneOrTheOther(x == 1))
  println(oneOrTheOther(x == 2))
}
/* Výstup:
To je pravda!
```

```
False
*/
```

Místo dvou výrazů v `trueOrFalse()` je `oneOrTheOther()` jediný výraz. Výsledek tohoto výrazu je výsledkem funkce, takže výraz `if` se stává tělem funkce.

Šablony řetězců

Šablona `řetězce` je programový způsob generování `řetězce`.

Pokud před název identifikátoru vložíte znak `$`, šablona `String` vloží obsah tohoto identifikátoru do `řetězce String`:

```
// StringTemplates/StringTemplates.kt

fun main() {
  val answer = 42
  println('Nalezeno $odpověď!') // [1]
  println('tisk $1') // [2]
}
/* Výstup:
Nalezeno 42!
tisk 1 USD
*/
```

- **[1]** `$answer` nahrazuje hodnotu `answer`.
- **[2]** Pokud to, co následuje za `$`, není rozpoznatelné jako identifikátor programu, nic zvláštního se nestane.

Hodnoty můžete do `řetězce` vkládat také pomocí spojování (`+`):

```
// StringTemplates/StringConcatenation.kt

fun main() {
  val s = 'hi\n' // \n je znak nového řádku
```

```
  hodnota n = 11
  hodnota d = 3,14
  println('první: ' + s + 'druhý: ' +
    n + ', třetí: ' + d)
}
/* Výstup:
první: ahoj
druhý: 11, třetí: 3,14
*/
```

Umístěním výrazu uvnitř `${}` se výraz vyhodnotí. Návratová hodnota se převede na `řetězec` a vloží se do výsledného `řetězce`:

```
// StringTemplates/ExpressionInTemplate.kt

fun main() {
  val condition = true
  println(
    '${if (condition) 'a' else 'b'}')  // [1]
  hodnota x = 11
  println('$x + 4 = ${x + 4}')
}
/* Výstup:
a
11 + 4 = 15
*/
```

- **[1]** `if(podmínka) 'a' else 'b'` je vyhodnoceno a výsledek je nahrazen celým výrazem `${}`.

Pokud `řetězec` musí obsahovat speciální znak, například uvozovku, můžete tento znak escapovat pomocí `\` (*zpětného lomítka*) nebo použít literál `řetězce` v trojitých uvozovkách:

```
// StringTemplates/TripleQuotes.kt
```

```
fun main() {
  val s = 'value'
  println('s = \'$s\'.')
  println('''s = '$s'.''')
}
/* Výstup:
s = 'value'.
s = 'value'.
*/
```

Pomocí trojitých uvozovek vkládáte hodnotu výrazu stejným způsobem jako u `řetězce s` jednoduchými uvozovkami.

Typy čísel

Různé typy čísel se ukládají různými způsoby.

Pokud vytvoříte identifikátor a přiřadíte mu hodnotu celého čísla, Kotlin odvodí typ `Int`:

```
// NumberTypes/InferInt.kt

fun main() {
  val million = 1_000_000 // Infers Int
  println(milion)
}
/* Výstup:
1000000
*/
```

Kvůli čitelnosti umožňuje Kotlin v číselných hodnotách používat podtržítka.

Základní matematické operátory pro čísla jsou ty, které jsou k dispozici ve většině programovacích jazyků: sčítání (`+`), odčítání (`-`), dělení (`/`), násobení (`*`) a modul (`%`), který vytváří zbytek z celočíselného dělení:

```
// NumberTypes/Modulus.kt

fun main() {
  val čitatel: Int = 19
  val jmenovatel: Int = 10
  println(čitatel % jmenovatel)
}
/* Výstup:
9
*/
```

Celočíselné dělení zkracuje svůj výsledek:

```
// NumberTypes/IntDivisionTruncates.kt

fun main() {
  val čitatel: Int = 19
  val jmenovatel: Int = 10
  println(čitatel / jmenovatel)
}
/* Výstup:
1
*/
```

Pokud by operace výsledek zaokrouhlila, výsledek by byl `2`.

Pořadí operací se řídí základní aritmetikou:

```
// NumberTypes/OpOrder.kt

fun main() {
  println(45 + 5 * 6)
}
/* Výstup:
75
```

```
*/
```

Nejprve se provede operace násobení `5 * 6` a poté sčítání `45 + 30`.

Pokud chcete, aby se nejprve vyskytlo `45 + 5`, použijte závorky:

```
// NumberTypes/OpOrderParens.kt

fun main() {
  println((45 + 5) * 6)
}
/* Výstup:
300
*/
```

Nyní vypočítáme *index tělesné hmotnosti* (BMI), což je hmotnost v kilogramech dělená druhou mocninou výšky v metrech. Pokud je váš BMI nižší než `18,5`, máte podváhu. V rozmezí `18,5` až `24,9` se jedná o normální hmotnost. BMI `25` a vyšší je nadváha. Tento příklad také ukazuje preferovaný styl formátování, když se parametry funkce nevejdou na jeden řádek:

```
// NumberTypes/BMIMetric.kt

fun bmiMetric(
  hmotnost: Dvojnásobná,
  výška: Dvojitá
): String {
  val bmi = hmotnost / (výška * výška) // [1]
  return if (bmi < 18,5) 'Podváha'
    else if (bmi < 25) 'Normální hmotnost'
    jinak "Nadváha
}

fun main() {
  hodnota weight = 72.57 // 160 liber
```

```
  val height = 1.727 // 68 palců
  val status = bmiMetric(hmotnost, výška)
  println(status)
}
/* Výstup:
Normální hmotnost
*/
```

- **[1]** Pokud odstraníte závorky, vydělíte `hmotnost` `výškou` a výsledek vynásobíte `výškou`. To je mnohem větší číslo a špatná odpověď.

`Funkce bmiMetric()` používá pro hmotnost a výšku hodnotu Doubles. `Double` pojme velmi velká a velmi malá čísla s pohyblivou řádovou čárkou.

Zde je verze používající anglické jednotky reprezentované parametry `Int`:

```
// NumberTypes/BMIEnglish.kt

fun bmiEnglish(
  hmotnost: Int,
  výška: Int
): String {
  val bmi =
    hmotnost / (výška * výška) * 703,07 // [1]
  return if (bmi < 18,5) 'Podváha'
    else if (bmi < 25) 'Normální hmotnost'
    jinak "Nadváha
}

fun main() {
  hodnota weight = 160
  hodnota height = 68
```

```
  val status = bmiEnglish(hmotnost, výška)
  println(status)
}
/* Výstup:
Podváha
*/
```

Proč se výsledek liší od `funkce bmiMetric()`, která používá funkci Doubles? Když dělíte celé číslo jiným celým číslem, Kotlin vytvoří celočíselný výsledek. Standardní způsob, jak se vypořádat se zbytkem při celočíselném dělení, je *zkrácení,* což znamená "useknout a zahodit" (nedochází k zaokrouhlování). Takže když vydělíte `5 dvěma`, dostanete `2` a `7/10` je nula. Když Kotlin počítá `bmi` ve výrazu **[1]**, vydělí `160 68 * 68` a dostane nulu. Poté vynásobí nulu číslem `703,07` a získá nulu.

Chcete-li se tomuto problému vyhnout, přesuňte položku `703.07` na začátek výpočtu. Výpočty jsou pak nuceny být `dvojnásobné`:

```
val bmi = 703,07 * hmotnost / (výška * výška)
```

Tomuto problému zabraňují parametry `Double` v `bmiMetric()`. Převeďte výpočty na požadovaný typ co nejdříve, abyste zachovali přesnost.

Všechny programovací jazyky mají omezení, co mohou uložit v rámci celého čísla. Typ `Int v` jazyce Kotlin může nabývat hodnot v rozmezí -2^{31} až $+2^{31}$ -1, což je omezení 32bitové reprezentace `Int`. Pokud sečtete nebo vynásobíte dva dostatečně velké Inty, dojde k přetečení výsledku:

```
// NumberTypes/IntegerOverflow.kt

fun main() {
  val i: Int = Int.MAX_VALUE
  println(i + i)
}
/* Výstup:
-2
```

```
*/
```

`Int.MAX_VALUE` je předdefinovaná hodnota, která je největším číslem, které může `Int` obsahovat.

Výsledkem přetečení je zjevně nesprávný výsledek, protože je záporný a mnohem menší, než jsme očekávali. Kotlin vydá varování, kdykoli zjistí potenciální přetečení.

Zabránit přetečení je vaší odpovědností jako vývojáře. Kotlin nedokáže vždy odhalit přetečení během kompilace a nezabrání mu, protože by to mělo nepřijatelný dopad na výkon.

Pokud váš program obsahuje velká čísla, můžete použít Longs, které pojmou hodnoty od -2^{63} do $+2^{63}$ -1. Chcete-li definovat `val` typu `Long`, můžete tento typ zadat explicitně nebo na konec číselného literálu vložit `L`, čímž Kotlinu řeknete, aby s touto hodnotou zacházel jako s `Long`:

```
// NumberTypes/LongConstants.kt

fun main() {
  hodnota i = 0 // Infers Int
  hodnota l1 = 0L // L vytváří Long
  val l2: Long = 0 // Explicitní typ
  println('$l1 $l2')
}
/* Výstup:
0 0
*/
```

Změnou na Longs zabráníme přetečení v `IntegerOverflow.kt`:

```
// NumberTypes/UsingLongs.kt

fun main() {
  val i = Int.MAX_VALUE
  println(0L + i + i) // [1]
```

```
  println(1_000_000 * 1_000_000L) // [2]
}
/* Výstup:
4294967294
1000000000000
*/
```

Použití číselného literálu v **[1]** i **[2]** vynucuje výpočty typu `Long` a také vytváří výsledek typu `Long`. Na místě, kde se písmeno `L` objeví, nezáleží. Pokud je jedna z hodnot `Long`, výsledný výraz je `Long`.

Přestože mohou obsahovat mnohem větší hodnoty než Inty, mají Longy stále omezení velikosti:

```
// NumberTypes/BiggestLong.kt

fun main() {
  println(Long.MAX_VALUE)
}
/* Výstup:
9223372036854775807
*/
```

`Long.MAX_VALUE` je největší hodnota, kterou může `Long` obsahovat.

Booleans

`if` Výrazy prokázaly operátor "not" `!`, který neguje `logickou` hodnotu. Tento atom zavádí další *booleovskou algebru*.

Začneme operátory "a" a "nebo":

- `&&` (a): a &: dává `pravdu` pouze tehdy, pokud je `pravdivý` jak `logický` výraz vlevo od operátoru, tak výraz vpravo od něj.
- `||` (nebo): Pokud je `pravdivý` buď výraz vlevo, nebo vpravo od operátoru, nebo pokud jsou `pravdivé` oba.

V tomto příkladu určíme, zda je podnik otevřený nebo zavřený, na základě `hodiny`:

```
// Booleans/Open1.kt

fun isOpen1(hour: Int) {
  hodnota open = 9
  hodnota closed = 20
  println('Provozní doba: $open - $closed')
  val status =
    if (hour >= open && hour <= closed) // [1]
      Pravda
    jinak
      false
  println('Otevřeno: $status')
}

fun main() = isOpen1(6)
/* Výstup:
Provozní doba: 9 - 20
Otevřeno: false
*/
```

`main()` je volání jedné funkce, takže můžeme použít tělo výrazu, jak je popsáno v části Funkce.

Výraz `if` v **[1]** kontroluje, zda je `hodina` mezi otevírací dobou a zavírací dobou, takže kombinujeme výrazy pomocí `logického &&` (and).

Výraz `if` lze zjednodušit. Výsledkem výrazu `if(cond) true else false` je právě `cond`:

```
// Booleans/Open2.kt

fun isOpen2(hour: Int) {
```

```
  hodnota open = 9
  hodnota closed = 20
  println('Provozní doba: $open - $closed')
  val status = hour >= open && hour <= closed
  println('Otevřeno: $status')
}

fun main() = isOpen2(6)
/* Výstup:
Provozní doba: 9 - 20
Otevřeno: false
*/
```

Obraťme logiku a zkontrolujme, zda je obchod aktuálně uzavřen. Operátor "nebo" `||` dává `hodnotu true`, pokud je splněna alespoň jedna z podmínek:

```
// Booleans/Closed.kt

fun isClosed(hour: Int) {
  hodnota open = 9
  hodnota closed = 20
  println('Provozní doba: $open - $closed')
  val status = hour < open || hour > closed
  println('Uzavřeno: $status')
}

fun main() = isClosed(6)
/* Výstup:
Provozní doba: 9 - 20
Uzavřeno: true
*/
```

Booleovské operátory umožňují složitou logiku v kompaktních výrazech. Snadno se však mohou stát nepřehlednými. Snažte se o srozumitelnost a explicitně specifikujte své záměry.

Zde je příklad složitého logického výrazu, u kterého různé pořadí vyhodnocení vede k různým výsledkům:

```
// Booleans/EvaluationOrder.kt

fun main() {
  val sunny = true
  val hoursSleep = 6
  val exercise = false
  hodnota temp = 55

  // [1]:
  hodnota happy1 = sunny && temp > 50 ||
    exercise && hoursSleep > 7
  println(happy1)

  // [2]:
  val sameHappy1 = (sunny && temp > 50) ||
    (exercise && hoursSleep > 7)
  println(sameHappy1)

  // [3]:
  val notSame =
    (slunečno && temp > 50 || cvičení) &&
      hoursSleep > 7
  println(notSame)
}
/* Výstup:
Pravda
```

```
Pravda
false
*/
```

`Booleovské` výrazy jsou `slunečno`, `teplota > 50`, `cvičení` a `hodinspánku > 7`. Výraz `happy1` čteme jako "Je slunečno *a* teplota je vyšší než 50 *nebo* jsem cvičil *a* spal více než 7 hodin.". Má však `&&` přednost před `||`, nebo naopak?

Výraz v **[1]** používá výchozí pořadí vyhodnocování jazyka Kotlin. Výsledek je stejný jako u výrazu v **[2]**, protože bez závorek se nejprve vyhodnotí "a" a teprve potom "nebo". Výraz v **[3]** používá závorky, které vedou k jinému výsledku. Ve výrazu **[3]** jsme šťastní pouze tehdy, když spíme alespoň 7 hodin.

Opakování pomocí `while`

Počítače jsou ideální pro opakující se úkoly.

Nejzákladnější forma opakování používá klíčové slovo `while`. To opakuje blok tak dlouho, dokud je řídicí *logický výraz* `pravdivý`:

```
while (logický výraz) {
  // Kód, který se má opakovat
}
```

Logický výraz je vyhodnocen jednou na začátku smyčky a znovu před každou další iterací v bloku.

```
// RepetitionWithWhile/WhileLoop.kt

fun condition(i: Int) = i < 100 // [1]

fun main() {
  var i = 0
  while (condition(i)) { // [2]
    print('.')
```

```
    i += 10 // [3]
  }
}
/* Výstup:
..........
*/
```

- **[1]** Operátor porovnání `<` vytváří výsledek typu `Boolean`, takže Kotlin odvozuje typ výsledku pro `condition()` jako `Boolean`.
- **[2]** Podmíněný výraz pro `while` říká: "opakuj příkazy v těle, dokud funkce `condition()` vrátí `true`."
- **[3]** Operátor `+=` přičte `10` k `i` a výsledek přiřadí `i` v rámci jedné operace (aby to fungovalo, musí být `i` `var`). To je ekvivalentní:

```
i = i + 10
```

Existuje i druhý způsob použití příkazu `while` ve spojení s klíčovým slovem `do`:

```
proveďte {
  // Kód, který se má opakovat
} while (logický výraz)
```

Přepsání `WhileLoop.kt` pro použití `do-while` produkuje:

```
// RepetitionWithWhile/DoWhileLoop.kt

fun main() {
  var i = 0
  proveďte {
    print('.')
    i += 10
  } while (condition(i))
}
/* Výstup:
..........
*/
```

Jediný rozdíl mezi `while` a `do-while` spočívá v tom, že tělo `do-while` se vždy provede alespoň jednou, i když logický výraz zpočátku dává `hodnotu false`. V `while`, pokud je podmínka poprvé `nepravdivá`, se tělo nikdy neprovede. V praxi se `do-while` používá méně často než `while`.

Zkrácené verze operátorů přiřazení jsou k dispozici pro všechny aritmetické operace: +=Zde se používají `-=` a `%=`:

```
// RepetitionWithWhile/AssignmentOperators.kt

fun main() {
  var n = 10
  hodnota d = 3
  print(n)
  while (n > d) {
    n -= d
    print(' - $d')
  }
  println(' = $n')

  var m = 10
  print(m)
  m %= d
  println(' % $d = $m')
}
/* Výstup:
10 - 3 - 3 - 3 = 1
10 % 3 = 1
*/
```

Výpočet zbytku celočíselného dělení dvou přirozených čísel začneme cyklem `while` a poté použijeme operátor zbytku.

Sčítání `1` a odčítání `1` od čísla jsou tak časté, že mají vlastní operátory inkrementace a dekrementace: `++` a `--`. Operátor `i += 1` můžete nahradit operátorem `i++`:

```
// RepetitionWithWhile/IncrementOperator.kt

fun main() {
  var i = 0
  while (i < 4) {
    print('.')
    i++
  }
}
/* Výstup:
....
*/
```

V praxi se cykly `while` nepoužívají pro iteraci nad rozsahem čísel. Místo toho se používá smyčka `for`. Tomu se věnujeme v dalším atomu.

Smyčky a rozsahy

Klíčové slovo `for` spustí blok kódu pro každou hodnotu v posloupnosti.

Množina hodnot může být rozsah celých čísel, `řetězec` nebo, jak uvidíte později v knize, kolekce položek. Klíčové slovo `in` označuje, že procházíte hodnoty:

```
for (v in values) {
  // Udělejte něco s v
}
```

Při každém průchodu smyčkou je `v` zadán další prvek v `hodnotách`.

Zde je smyčka `for`, která opakuje určitý počet akcí:

```
// LoopingAndRanges/RepeatThreeTimes.kt
```

```
fun main() {
  for (i in 1..3) {
    println('Hey $i!')
  }
}
/* Výstup:
Hej, jedničko!
Ahoj 2!
Ahoj 3!
*/
```

Na výstupu se zobrazí index `i`, který obdrží každou hodnotu v rozsahu 1 až 3.

Rozsah je interval hodnot definovaný dvojicí koncových bodů. Rozsahy lze definovat dvěma základními způsoby:

```
// LoopingAndRanges/DefiningRanges.kt

fun main() {
  val range1 = 1..10 // [1]
  val range2 = 0 až 10 // [2]
  println(range1)
  println(range2)
}
/* Výstup:
1..10
0..9
*/
```

- **[1]** Syntaxe `..` zahrnuje obě hranice výsledného rozsahu.
- **[2]** `až do` vyloučení konce. Výstup ukazuje, že `10` není součástí rozsahu.

Zobrazení rozsahu vytváří čitelný formát.

Tím se sečtou čísla od 10 do 100:

```
// LoopingAndRanges/SumUsingRange.kt

fun main() {
  var sum = 0
  for (n v 10..100) {
    sum += n
  }
  println('sum = $sum')
}
/* Výstup:
součet = 5005
*/
```

Rozsah můžete iterovat v opačném pořadí. Můžete také použít hodnotu `kroku`, abyste změnili interval z výchozí hodnoty `1`:

```
// LoopingAndRanges/ForWithRanges.kt

fun showRange(r: IntProgression) {
  for (i in r) {
    print('$i ')
  }
  print(' // $r')
  println()
}

fun main() {
  showRange(1..5)
  showRange(0 až 5)
  showRange(5 downTo 1) // [1]
```

```
  showRange(0..9 krok 2) // [2]
  showRange(0 až 10 krok 3) // [3]
  showRange(9 downTo 2 krok 3)
}
/* Výstup:
1 2 3 4 5 // 1..5
0 1 2 3 4 // 0..4
5 4 3 2 1 // 5 downTo 1 krok 1
0 2 4 6 8 // 0..8 krok 2
0 3 6 9 // 0..9 krok 3
9 6 3 // 9 downTo 3 krok 3
*/
```

- **[1]** `downTo` vytváří klesající rozsah.
- **[2]** `krok` mění interval. Zde se interval krokuje o hodnotu dvě místo o jednu.
- **[3]** `dokud` lze použít i `krokování`. Všimněte si, jaký to má vliv na výstup.

V každém případě tvoří posloupnost čísel aritmetickou progresi. `showRange()` přijímá parametr `IntProgression`, což je vestavěný typ, který zahrnuje rozsahy `Int`. Všimněte si, že `řetězcová` reprezentace každé `IntProgression`, jak se objevuje ve výstupním komentáři pro každý řádek, se často liší od rozsahu předaného do funkce `showRange()` - `IntProgression` převádí vstup do ekvivalentní běžné podoby.

Můžete také vytvořit řadu znaků. Tento `for` iteruje od `a` do `z`:

```
// LoopingAndRanges/ForWithCharRange.kt

fun main() {
  for (c in 'a'.'z') {
    print(c)
  }
}
```

```
/* Výstup:
abcdefghijklmnopqrstuvwxyz
*/
```

Můžete iterovat nad rozsahem prvků, které jsou celými veličinami, jako jsou celá čísla a znaky, ale ne hodnoty s pohyblivou řádovou čárkou.

Hranaté závorky zpřístupňují znaky podle indexu. Protože znaky v `řetězci` začínáme počítat od nuly, `s[0]` vybere první znak `řetězce` `s`. Výběrem `s.lastIndex` získáme konečné číslo indexu:

```
// LoopingAndRanges/IndexIntoString.kt

fun main() {
  val s = 'abc'
  for (i in 0..s.lastIndex) {
    print(s[i] + 1)
  }
}
/* Výstup:
bcd
*/
```

Někdy se `s[0]` označuje jako "nultý znak".

Znaky jsou uloženy jako čísla odpovídající jejich kódům ASCII, takže přidáním celého čísla ke znaku vznikne nový znak odpovídající nové hodnotě kódu:

```
// LoopingAndRanges/AddingIntToChar.kt

fun main() {
  val ch: Char = 'a'
  println(ch + 25)
  println(ch < 'z')
}
```

```
/* Výstup:
z
Pravda
*/
```

Druhý příkaz `println()` ukazuje, že můžete porovnávat kódy znaků.

Cyklus `for` může iterovat nad řetězci přímo:

```
// LoopingAndRanges/IterateOverString.kt

fun main() {
  for (ch in 'Jnskhm ') {
    print(ch + 1)
  }
}
/* Výstup:
Kotlin!
*/
```

`ch` postupně přijímá jednotlivé znaky.

V následujícím příkladu funkce `hasChar()` iteruje `řetězec s` a testuje, zda obsahuje zadaný znak `ch`. `Návrat` uprostřed funkce ukončí funkci, když je nalezena odpověď:

```
// LoopingAndRanges/HasChar.kt

fun hasChar(s: String, ch: Char): {
  for (c in s) {
    if (c == ch) return true
  }
  vrátit false
}

fun main() {
```

```
  println(hasChar('kotlin', 't'))
  println(hasChar('kotlin', 'a'))
}
/* Výstup:
Pravda
false
*/
```

Další atom ukazuje, že `funkce hasChar()` je zbytečná - místo ní můžete použít vestavěnou syntaxi.

Pokud chcete akci jednoduše opakovat pevný počtekrát, můžete místo cyklu `for` použít funkci `repeat()`:

```
// LoopingAndRanges/RepeatHi.kt

fun main() {
  repeat(2) {
    println('ahoj!')
  }
}
/* Výstup:
Ahoj!
Ahoj!
*/
```

`repeat()` je funkce standardní knihovny, nikoli klíčové slovo. Jak byla vytvořena, se dozvíte mnohem později v knize.

The `v` Klíčové slovo

Klíčové slovo `in` testuje, zda se hodnota nachází v určitém rozsahu.

```
// InKeyword/MembershipInRange.kt

fun main() {
```

```
  val procent = 35
  println(procento v 1..100)
}
/* Výstup:
true
*/
```

V předmětu Booleans jste se naučili explicitně kontrolovat hranice:

```
// InKeyword/MembershipUsingBounds.kt

fun main() {
  val procent = 35
  println(0 <= procent && procent <= 100)
}
/* Výstup:
Pravda
*/
```

`0 <= x && x <= 100` je logicky ekvivalentní `x v 0..100`. IntelliJ IDEA navrhuje automaticky nahradit první tvar druhým, který je přehlednější a srozumitelnější.

Klíčové slovo `in` se používá pro iteraci i členství. Slovo `in` uvnitř řídicího výrazu cyklu `for` znamená iteraci, jinak `in` kontroluje příslušnost:

```
// InKeyword/IterationVsMembership.kt

fun main() {
  val values = 1..3
  for (v in values) {
    println('iterace $v')
  }
  hodnota v = 2
  if (v in values)
    println('$v je členem $values')
```

```
}
/* Výstup:
iterace 1
iterace 2
iterace 3
2 je členem 1..3
*/
```

Klíčové slovo `in` není omezeno na rozsahy. Můžete také zkontrolovat, zda je znak součástí `řetězce`. Následující příklad používá `in` místo `hasChar()` z předchozího atomu:

```
// InKeyword/InString.kt

fun main() {
  println('t' v 'kotlin')
  println('a' v 'kotlin')
}
/* Výstup:
Pravda
false
*/
```

Později v knize uvidíte, že to funguje i u jiných typů.

Zde se testuje, zda znak patří do rozsahu znaků:

```
// InKeyword/CharRange.kt

fun isDigit(ch: Char) = ch in '0'..'9'

fun notDigit(ch: Char) =
  ch !in '0'..'9' // [1]

fun main() {
  println(isDigit('a'))
```

```
  println(isDigit('5'))
  println(notDigit('z'))
}
/* Výstup:
false
Pravda
Pravda
*/
```

- **[1]** `!in` kontroluje, zda hodnota nepatří do rozsahu.

Můžete vytvořit `Dvojitý` rozsah, ale můžete jej použít pouze ke kontrole členství:

```
// InKeyword/FloatingPointRange.kt

fun inFloatRange(n: Double) {
  hodnota r = 1.0..10.0
  println('$n v $r? ${n v r}')
}

fun main() {
  inFloatRange(0.999999)
  inFloatRange(5.0)
  inFloatRange(10.0)
  inFloatRange(10.0000001)
}
/* Výstup:
0.999999 v 1.0..10.0? false
5.0 v 1.0..10.0? true
10.0 v 1.0..10.0? true
10.0000001 v 1.0..10.0? false
*/
```

Rozsahy s plovoucí desetinnou čárkou lze vytvořit pouze pomocí `..`, protože `dokud` by to znamenalo vyloučení čísla s plovoucí desetinnou čárkou jako koncového bodu, což nedává smysl.

Můžete zkontrolovat, zda je `řetězec` členem rozsahu řetězců:

```
// InKeyword/StringRange.kt

fun main() {
  println('ab' v 'aa'..'az')
  println('ba' v 'aa'..'az')
}
/* Výstup:
Pravda
false
*/
```

Kotlin zde používá abecední porovnávání.

Výrazy a výroky

Příkazy a *výrazy* jsou ve většině programovacích jazyků nejmenšími užitečnými částmi kódu.

Je tu základní rozdíl: výrok má účinek, ale nemá žádný výsledek. Výraz vždy vytváří výsledek.

Protože příkaz nevytváří výsledek, musí změnit stav svého okolí, aby byl užitečný. Jiný způsob, jak to říci, je "příkaz se volá kvůli svým *vedlejším účinkům*" (tj. kvůli tomu, co dělá *jiného* než produkuje výsledek). Jako pomůcka pro zapamatování:

Výpis mění stav.

Jedna z definic slova "expresní" je "vytlačit nebo vymačkat", jako například "vymačkat šťávu z pomeranče". Takže

Výraz vyjadřuje.

To znamená, že přináší výsledek.

Smyčka `for` je příkaz v jazyce Kotlin. Nemůžete ji přiřadit, protože nemá žádný výsledek:

```
// ExpressionsStatements/ForIsAStatement.kt

fun main() {
  // To nelze:
  // val f = for(i in 1..10) {}
  // Chybová zpráva překladače:
  // for není výraz a
  // zde jsou povoleny pouze výrazy
}
```

Pro jeho vedlejší účinky se používá smyčka `for`.

Výraz vytváří hodnotu, kterou lze přiřadit nebo použít jako součást jiného výrazu, zatímco příkaz je vždy prvkem nejvyšší úrovně.

Každé volání funkce je výraz. I když funkce vrací `jednotku` a je volána pouze pro své vedlejší efekty, výsledek lze přiřadit:

```
// ExpressionsStatements/UnitReturnType.kt

fun unitFun() = Unit

fun main() {
  println(unitFun())
  hodnota u1: Unit = println(42)
  println(u1)
  val u2 = println(0) // Odvození typu
  println(u2)
}
```

```
/* Výstup:
kotlin.Unit
42
kotlin.Unit
0
kotlin.Unit
*/
```

Typ `Unit` obsahuje jedinou hodnotu `Unit`, kterou můžete vrátit přímo, jak je vidět v příkazu `unitFun()`. Volání `funkce println()` rovněž vrací `jednotku Unit`. `Hodnota u1` zachycuje návratovou hodnotu funkce `println()` a je explicitně deklarována jako `Unit`, zatímco `u2` využívá typovou inferenci.

`if` vytvoří výraz, takže můžete přiřadit jeho výsledek:

```
// ExpressionsStatements/AssigningAnIf.kt

fun main() {
  val result1 = if (11 > 42) 9 else 5

  val result2 = if (1 < 2) {
    hodnota a = 11
    a + 42
  } jinak 42

  val result3 =
    if ('x' < 'y')
      println('x < y')
    jinak
      println('x > y')

  println(result1)
  println(result2)
```

```
  println(result3)
}
/* Výstup:
x < y
5
53
kotlin.Unit
*/
```

První výstupní řádek je `x < y`, přestože `výsledek3` je zobrazen až na konci `main()`. K tomu dochází proto, že vyhodnocení `result3` volá `println()` a k vyhodnocení dochází v okamžiku, kdy je definován `result3`.

Všimněte si, že `a` je definováno uvnitř bloku kódu pro `result2`. Výsledek posledního výrazu se stane výsledkem výrazu `if`; zde je to součet 11 a 42. Ale co `a`? Jakmile opustíte blok kódu (přesunete se mimo kudrnaté závorky), nemáte přístup k `a`. Je *dočasný* a po opuštění *oboru* tohoto bloku je zahozen.

Operátor přírůstku `i++` je také výraz, i když vypadá jako příkaz. Kotlin následuje přístup používaný jazyky podobnými jazyku C a poskytuje dvě verze operátorů inkrementace a dekrementace s mírně odlišnou sémantikou. Operátor prefix se objevuje před operandem, jako ve tvaru `++i`, a vrací hodnotu poté, co dojde k inkrementaci. Můžete jej číst jako "nejprve proveďte inkrementaci a pak vraťte výslednou hodnotu". Postfixový operátor je umístěn za operandem, jako v `i++`, a vrací hodnotu `i` předtím, než dojde k inkrementaci. Lze jej číst jako "nejprve vytvořte výsledek, pak proveďte inkrementaci".

```
// ExpressionsStatements/PostfixVsPrefix.kt

fun main() {
  i = 10
  println(i++)
  println(i)
```

```
  var j = 20
  println(++j)
  println(j)
}
/* Výstup:
10
11
21
21
*/
```

Operátor dekrementace má také dvě verze: `--i` a `i--`. Používání operátorů inkrementace a dekrementace v rámci jiných výrazů se nedoporučuje, protože může vést k matoucímu kódu:

```
// ExpressionsStatements/Confusing.kt

fun main() {
  var i = 1
  println(i++ + ++i)
}
```

Zkuste odhadnout, jaký bude výstup, a pak jej zkontrolujte.

Vytvořeno v rámci projektu: **DANTE**, reg. č. NPO_UPCE_MSMT-16591/2022

Financováno Evropskou unií

NextGenerationEU

Národní plán obnovy

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Mobilní aplikace

Kotlin - třídy, enums, pole

Popis hry

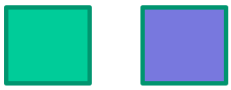
Budeme vytvářet jednoduchou RPG hru.

- Po omezené herní ploše se pohybuje postava hrdiny, který má zabít všechny nepřátelé. Hra končí, ve chvíli kdy jsou mrtvi všichni nepřátelé nebo hrdina.
- Herní políčka mají určitý terén. Na některé políčka může hrdina vstoupit (louka, most), na nějaké ne (les, řeka, hranice).
- Hrdina vidí pouze na sousední políčka.
- Hrdina má vlastnosti - zdraví, útok, obranu a léčení.
- Hrdina si své vlastnosti může vylepšovat sebráním předmětů. Předmět může hrdina sebrat, pokud stojí na stejném hracím poli, jako je předmět.
- Na hrací ploše jsou rozmístěny různí nepřátelé s různými vlastnostmi.
- Na nepřátele může hrdina zaútočit, pokud stojí na stejném hracím poli, jako je nepřítel.
- Hodnotícím kritériem je počet zahraných tahů.

Popis vytvářených tříd

GameField

Třída reprezentuje jedno hrací pole, které má mít určený terén. Můžeme si je přestavit jako různobarevné čtverce.



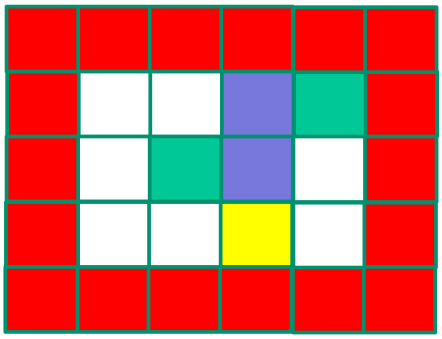
GamePlan

Třída reprezentuje herní plán. Velikost hracího plánu je určena na začátku hry. Interně bude reprezentována jako dvourozměrné pole herních polí.

Na okraji budou herní pole typu hranice, kam hrdina nesmí vkročit. Na náhodném sloupci bude řeka, která bude mít na náhodné pozici most. Most je jediné místo, kde lze řeka překonat.

Na herním plánu budou náhodně vygenerované lesy (viz úkol).

Zbylé políčka budou průchozí louky.



Třída GamePlan bude obsahovat metody pro generování herního plánu a metody pro zjišťování terénu, vykreslení mapy apod.

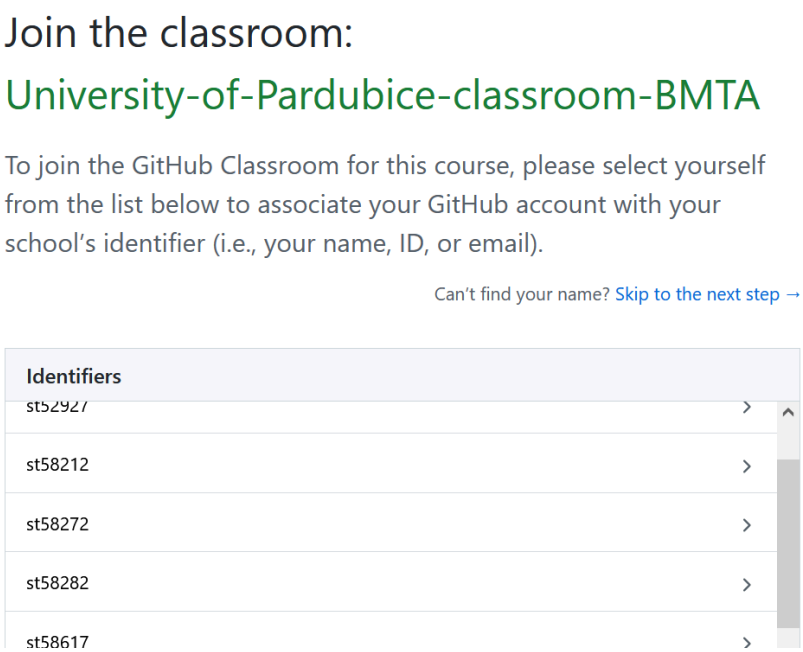
Game

Třída Game bude zastřešovat celou hru. Bude obsahovat herní plán a později hrdinu, nepřátelé a předměty.

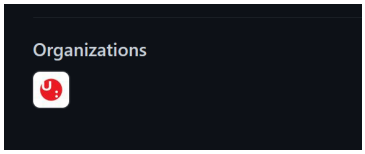
Přihlášení do gitu

Nejprve se připojíme do github classrom.

Zvolte svůj login a potvrďte svůj připojení. Na pozadí se vygeneruje repository.

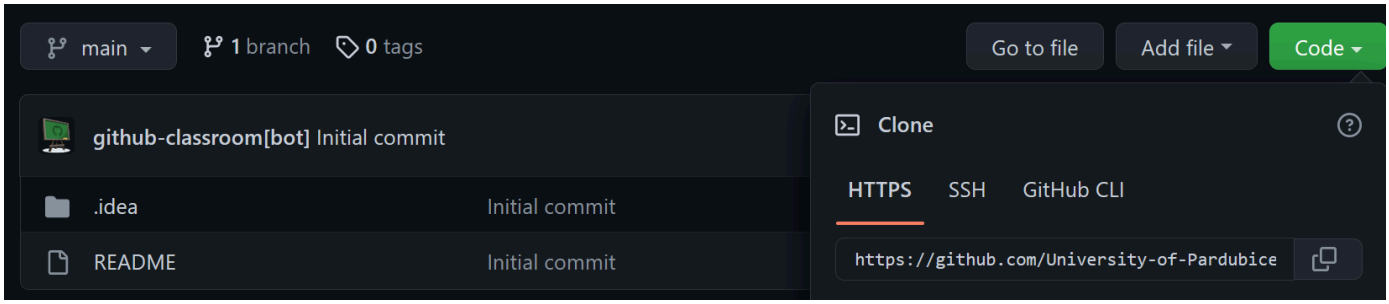


Když se přihlásíte na stránky www.github.com, tak byste měli vidět na hlavní stránce pod Organization ikonu UPCE.



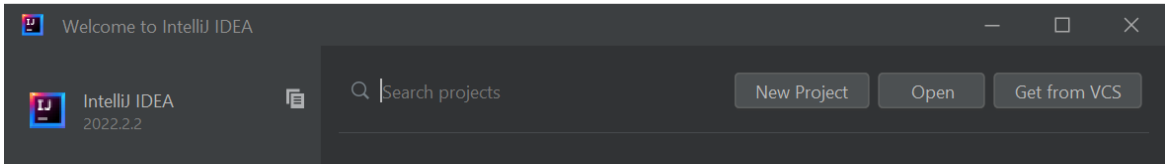
Kliknutím na ikonu se dostanete na seznam repository, kde byste měli vidět repository hra-kotlin-přihlašovací jméno.

V detailech repository bude pro nás důležitá adresa, kterou najdeme pod Code.

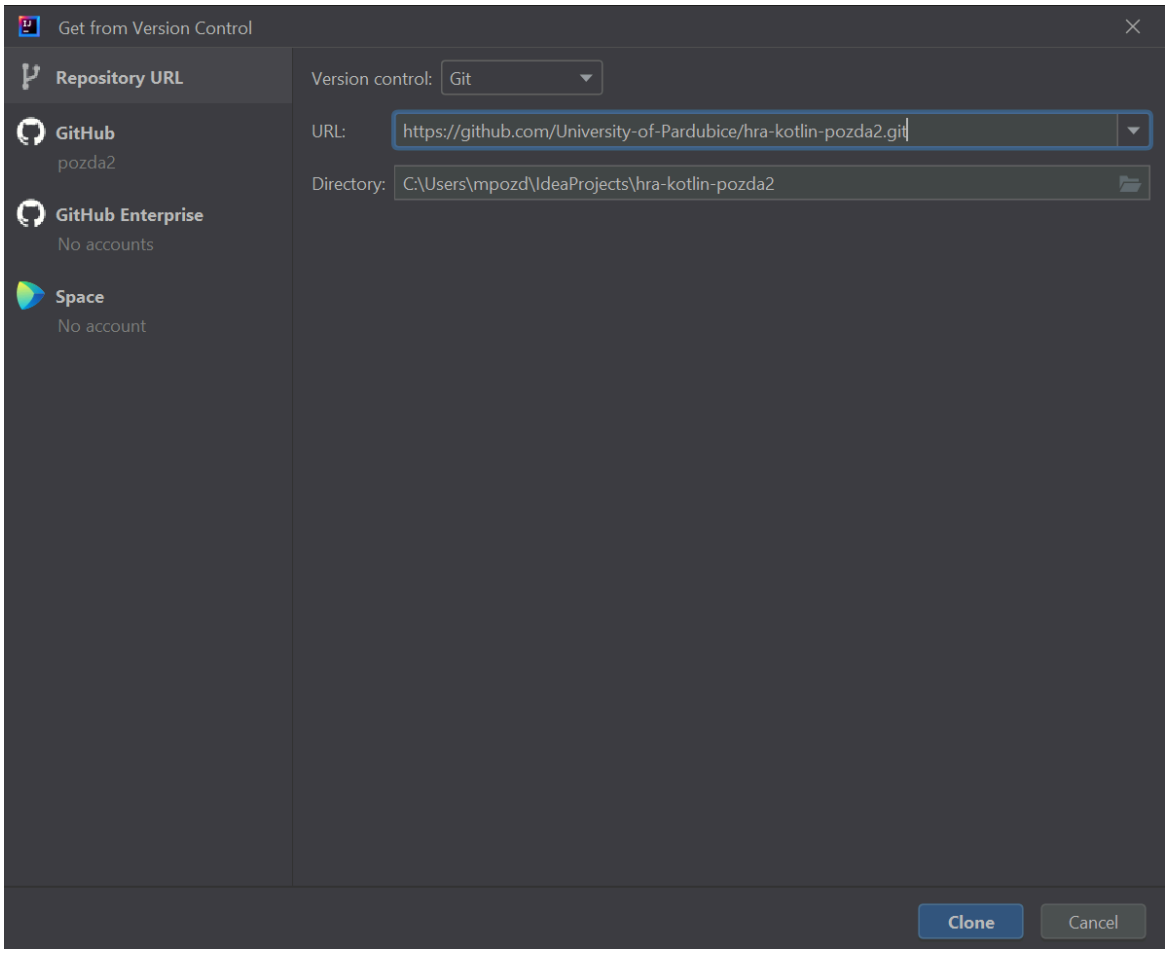


Založení projektu

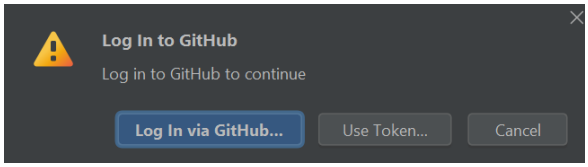
Projekt založíme naklonovaním vytvořeného repository Get from VCS



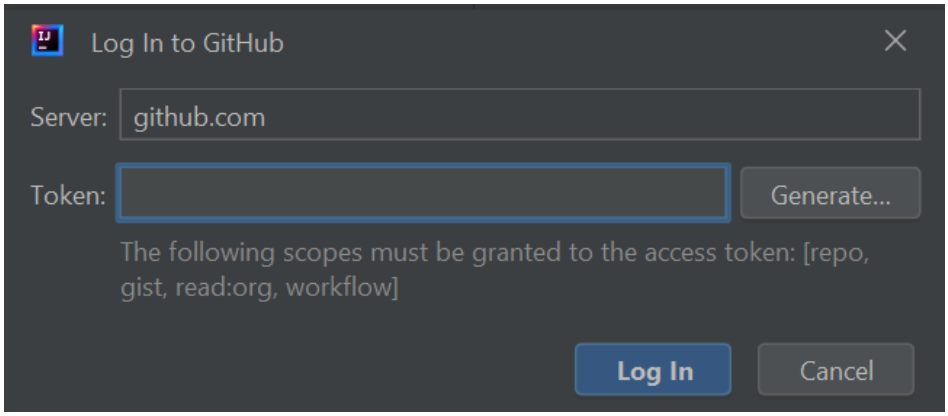
Do URL zadáme adresu naší repository ze stránky www.github.com



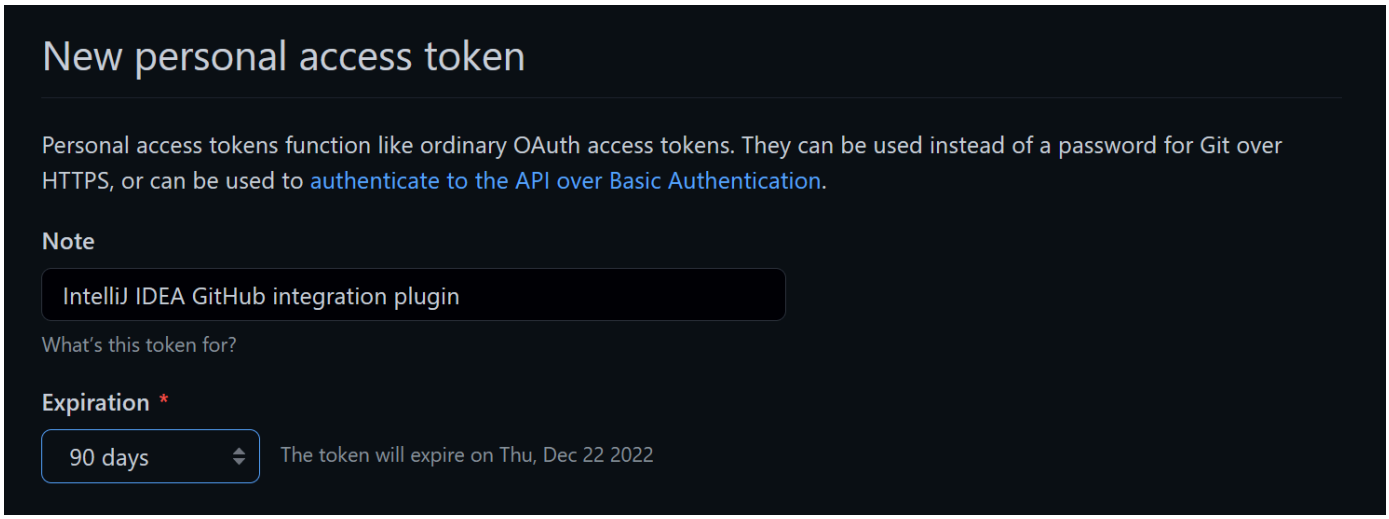
Protože zatím nemáme ověřený přístup mezi IntelliJ Idea a vzdáleným repository, zvolíme Log In via Github... v otevřeném oknu prohlížeče přístup autorizujeme.



Pokud tento postup z nějakého důvodu nebude fungovat, může použít Use Token. Kliknutím na Generate bude přesměrování na stránku Settings/Developer Setttings/Personal Access Token.



Nový token pojmenujte, zvolte správnou dobu expirace a dole zvolte generate Token.

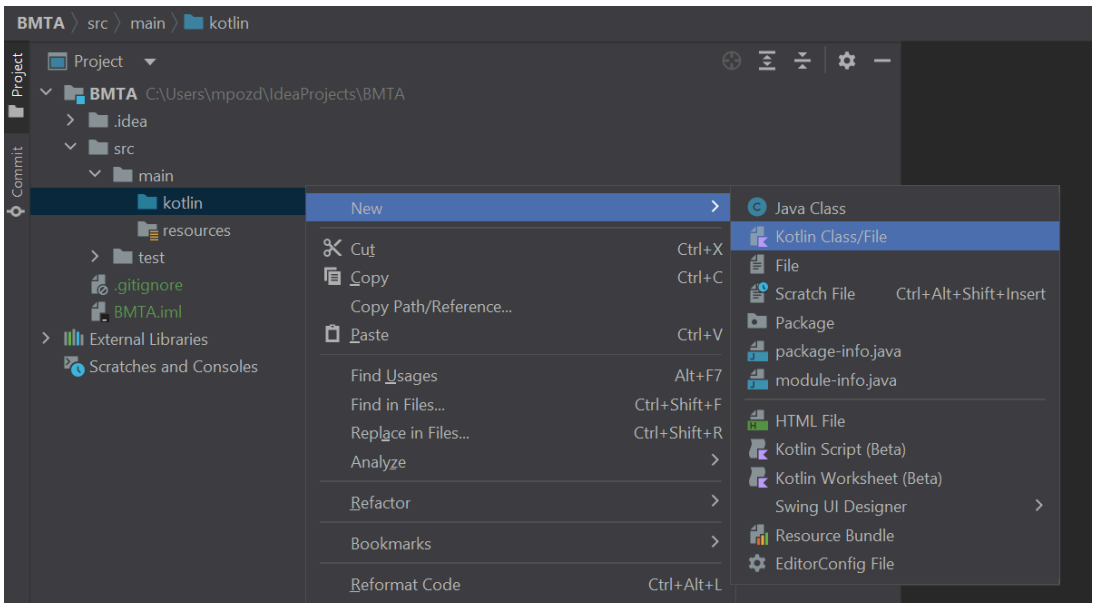


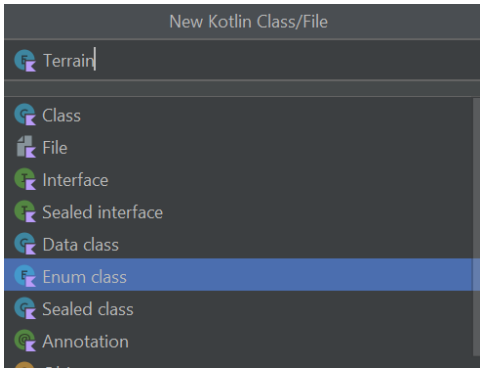
Pak by mělo dojít k naklonování projektu a my můžeme začít programovat.

Terén

Herní pole mohou mít pouze určité terény. Vytvoříme enum třídu, kde je definujeme.

Vytvoření nového kotlin souboru.





Základní terény jsou hranice, louka, les, řeka a most.

```
enum class Terrain {
    BORDER,
    MEADOW,
    FOREST,
    RIVER,
    BRIDGE
}
```

K hodnotám můžeme přiřadit dodatečné vlastnosti. My si přidáme slovní popis a znak, kterým bude terén reprezentovaný v textovém výpisu.

```
enum class Terrain (val description: String, val terrainChar: Char) {
    BORDER ("hranice", '#'),
    MEADOW ("louka", ' '),
    FOREST ("les", '|'),
    RIVER ("řeka", '*'),
    BRIDGE ("most", '=')
}
```

Herní Pole

Vytvoříme novou kotlin třídu GameField. Třída bude mít jediný atribut - terén, který bude typu Terrain. Třídu vytvoříme jako data class. Tím nám kotlin na pozadí vygeneruje konstruktor, settry, gettry a funkci equals, aniž bychom napsali jediný řádek.

```
data class GameField (var terrain: Terrain) {
}
```

Do třídy přidáme metodu isWalkable, která nám vrátí, zda na dané políčko hrdina může vstoupit. Metoda vrací hodnotu Boolean. V metodě používáme when, což je ekvivalent javového switch.

```
fun isWalkable () : Boolean {
    return when (this.terrain) {
        Terrain.BRIDGE, Terrain.MEADOW -> true
        Terrain.BORDER, Terrain.FOREST, Terrain.RIVER -> false
    }
}
```

Pozice

Hrdina, předměty, nepřátele, most apod. mají nějakou pozici na mapě určenou souřadnicemi x, y. Kotlin stejně jako Java neumí jednoduše vracet více hodnot z metody. Jednou z možností, jak toto obejít, jak vytvoření třídy Position, která spojí souřadnice x a y do jednoho objektu. To se nám bude hodit později.

Vytvoříme data class Position, která bude mít pouze dva atributy x, y typu Int. Na rozdíl od Javy se nepoužívá základní datových typ int, ale Int, což je třída.

```
data class Position (var x: Int, var y: Int) {
}
```

Herní Plán

Vytvoříme novou data class GamePlan. Třída bude mít atributy width a height, které určují velikost herního plánu. Dalším atributem bude numForrest tedy počet lesů, které bude herní plán obsahovat.

```
data class GamePlan (val width: Int = 20, val height : Int = 10, val numForrest: Int = 4) {
}
```

Hlavní atributem bude gamePlan tedy dvourozměrné pole typu GameField. Při vytváření ho rovnou naplníme samými loukami. Pro ukázku ho můžeme vytvořit jako private, takže pole nebude přístupné z jiných objektů, ale pouze přes metody.

```
private var gamePlan = Array(height) { Array(width) { GameField(Terrain.MEADOW) } }
```

Kotlin napozadí automaticky generuje konstruktor se 3 vstupními parametry. Konstruktor vytvoří gamePlan, který je zaplněn samými loukami. Do hry chceme určitou náhodu, takže herní pole během vytváření upravíme. To lze provést v bloku init, který se spouští při vytváření instance objektu. V něm zavoláme funkci generateGamePlan.

```
init {
    generateGamePlan()
}
```

Vytvoříme metodu generateGamePlan, ve které zavoláme funkci na generování hranic a pak řeky.

```
private fun generateGamePlan() {
        generateBorder()
        generateRiver()
    }
```

Implementace například může vypadat takto. Ale neefektivně procházíme všechna herní pole a když je index hraniční nastavíme terén. Zkuste kód upravit, aby byl více efektivní a neprocházel všechna pole.

```
private fun generateBorder() {
    for (i in 0 until height) {
        for (j in 0 until width) {
            if (i == 0 || j == 0 || i == height - 1 || j == width - 1) {
                gamePlan[i][j] = GameField(Terrain.BORDER)
            }
        }
    }
}
```

Dále budeme chtít vygenerovat náhodnou řeku. Pro to a pozdější použití si vytvoříme metodu generateRandomCoord, která vygeneruje náhodnou pozici x nebo y. Generují se čísla od 1 do size-1, protože nechceme generovat pozici na hranici. Metoda generateRandomPosition vratí náhodnout pozici. Protože budeme používat Random.nextInt, musí package importovat.

```
import kotlin.random.Random
```

```
private fun generateRandomCoord(size: Int): Int {
    return Random.nextInt(1, size -1)
}

private fun generateRandomPosition() : Position {
    return Position(generateRandomCoord(width), generateRandomCoord(height))
}
```

Když máme připravené metody pro generování náhodné pozice, můžeme doplnit funkci na vytvoření řeky.

```
private fun generateRiver() {
    val bridgePosition = generateRandomPosition()
    for (i in 1..height - 2) {
         gamePlan[i][bridgePosition.x] = GameField(Terrain.RIVER)
    }
    gamePlan[bridgePosition.y][bridgePosition.x] = GameField(Terrain.BRIDGE)
}
```

Protože jsme definovali gamePlan jako private, budeme potřebovat metodu, která nám minimálně vrátí herní políčko na pozici. Kdybychom nechali gamePlan otevřený, tak by tato metoda nebyla potřeba, ale kdokoliv by nám mohl měnit herní plán. Například pokud by zrušil hranice, hrdina by se mohl dostat mimo herní plán a způsobit null pointer exception ;-)

```
fun getGameField(position: Position): GameField {
    return gamePlan[position.y][position.x]
}
```

Poslední dnes implementovanou metodou bude map(). Jedná se o metodu, která nám pro kontrolu vykreslí vygenerovaný herní plán.

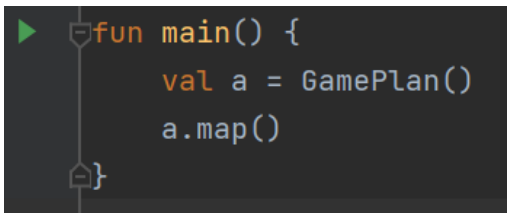
```
fun map() {
    for (i in 0 until height) {
        for (j in 0 until width) {
            print (gamePlan[i][j].terrain.terrainChar + " ")
        }
        println()
    }
}
```

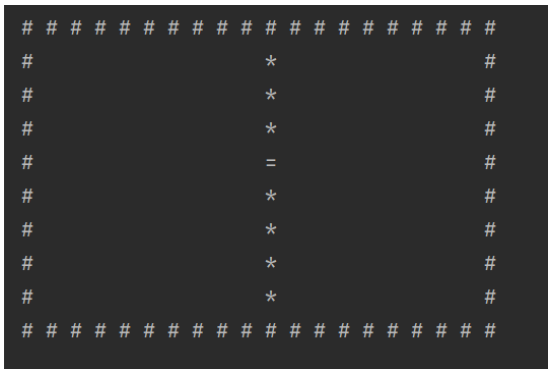
Hra

V souboru Main.kt Upravíme pouze metodu main, ve které si necháme vygenerovat herní plán a ten vypíšeme.

```
fun main() {
    val a = GamePlan()
    a.map()
}
```

Po spuštění, kliknutím na zelený trojúhelník, byste měli dostat podobný výstup.



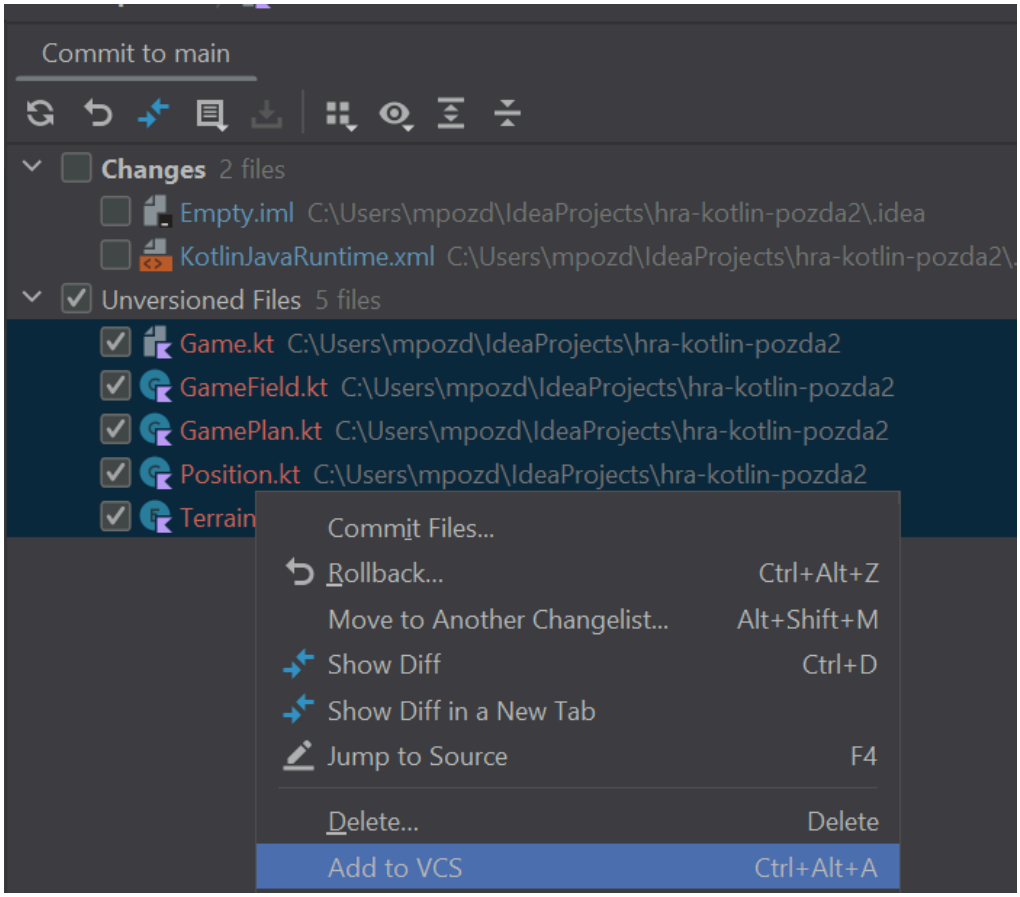


Voláme konstruktor s implicitními rozměry. Vyzkoušejte si generovat i jinak velké plány.

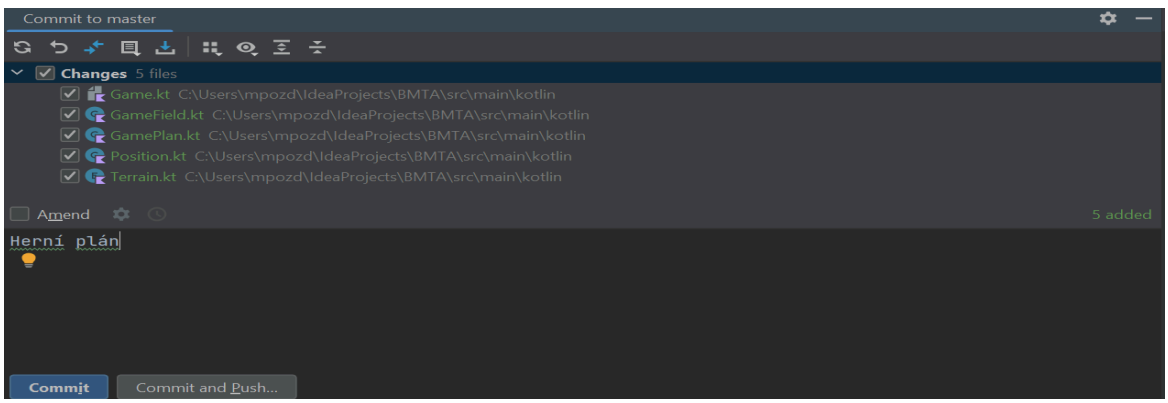
Git

Pokud funguje všechno, jak má, projekt uložíme do gitu. V IntelliJ se přepneme se do pohled pro Git (záložka Commit). A budeme muset vybrat soubory, které budeme chtít zahrnout do gitu a které. V závislosti na nastavení prostředí se mohlo stát, že nově vytvořené soubory se automaticky přidávají do Stage area (git add).

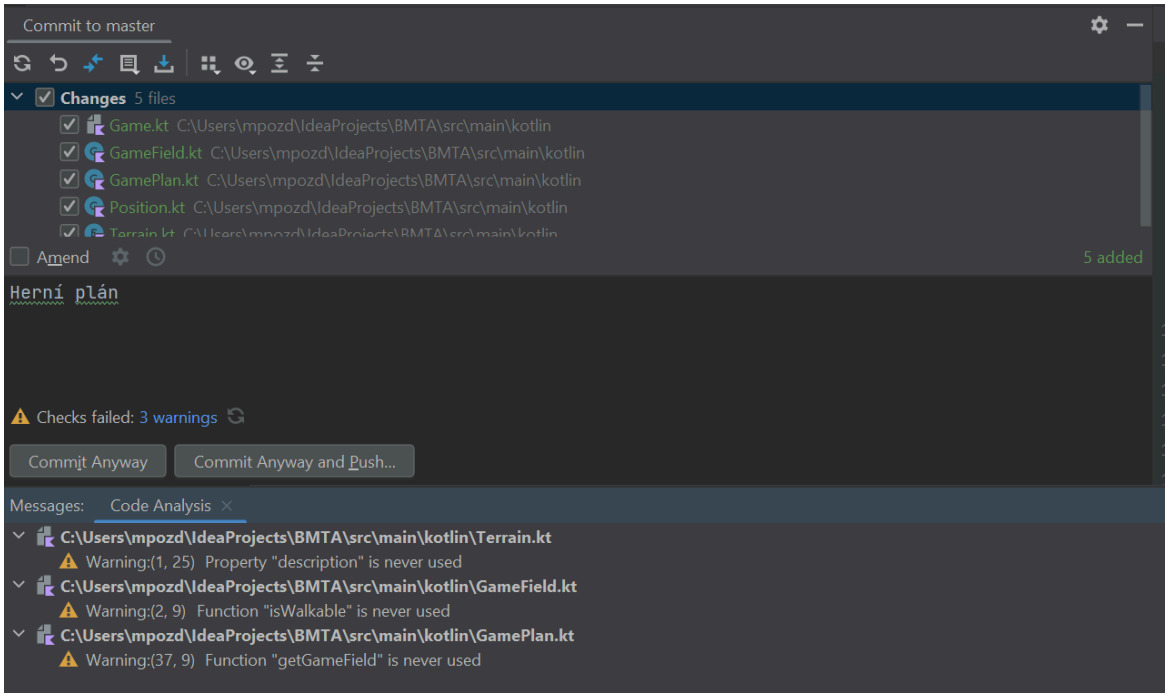
Do gitu přidáme nové soubory



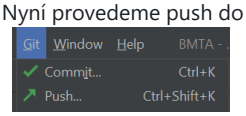
Pak již můžeme provést Commit.



V závislosti na nastavení IntelliJ Idea provede kontrolu projektu a zobrazí chyby, varování doporučení. V našem případě objevila nepoužité části kódu. My je budeme používat v budoucnu, takže Commit anyway.



Nyní provedeme push do vzdálené repository.



Vytvořeno v rámci projektu: **DANTE**, reg. č. NPO_UPCE_MSMT-16591/2022

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Naposledy změněno: čtvrtek, 2. května 2024, 15.44

◄ Kotlin - třídy, enums, pole - video

Přejít na...

Cvičení - úkol (generování lesa) ►

Mobilní aplikace

Kotlin - arrayList - text

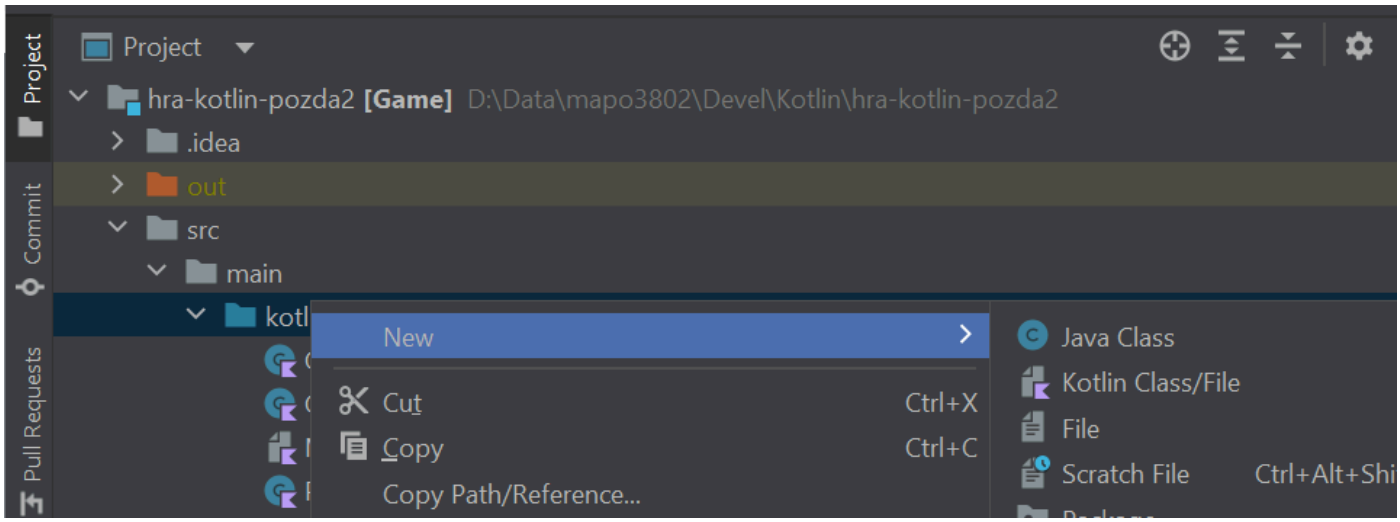
Cvičení - Hrdina

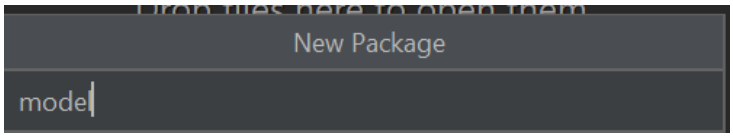
Do hry přidáme hrdinu, který bude mít určité vlastnost (zdraví, sílu útoku, sílu obrany, uzdravování) a zároveň bude schopen se pohybovat po herním plánu.

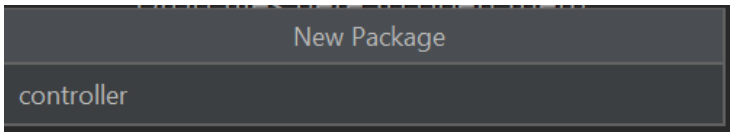
Rozšíříme hru, aby začala s hráčem interagovat. Bude vypisovat aktuální stav hry a bude od hráče přijímat pokyny.

Packages

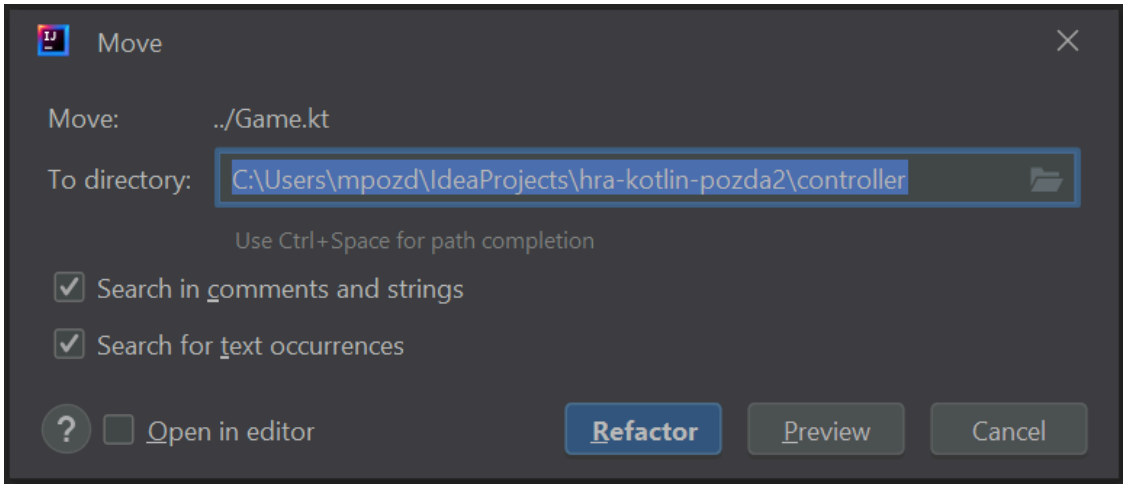
Ve vytvářených souborech si začneme dělat pořádek. Vytvoříme package model, do kterého budeme dávat třídy, které uchovávají stav objektů. Druhý package bude controller, který bude obsahovat logiku hry.



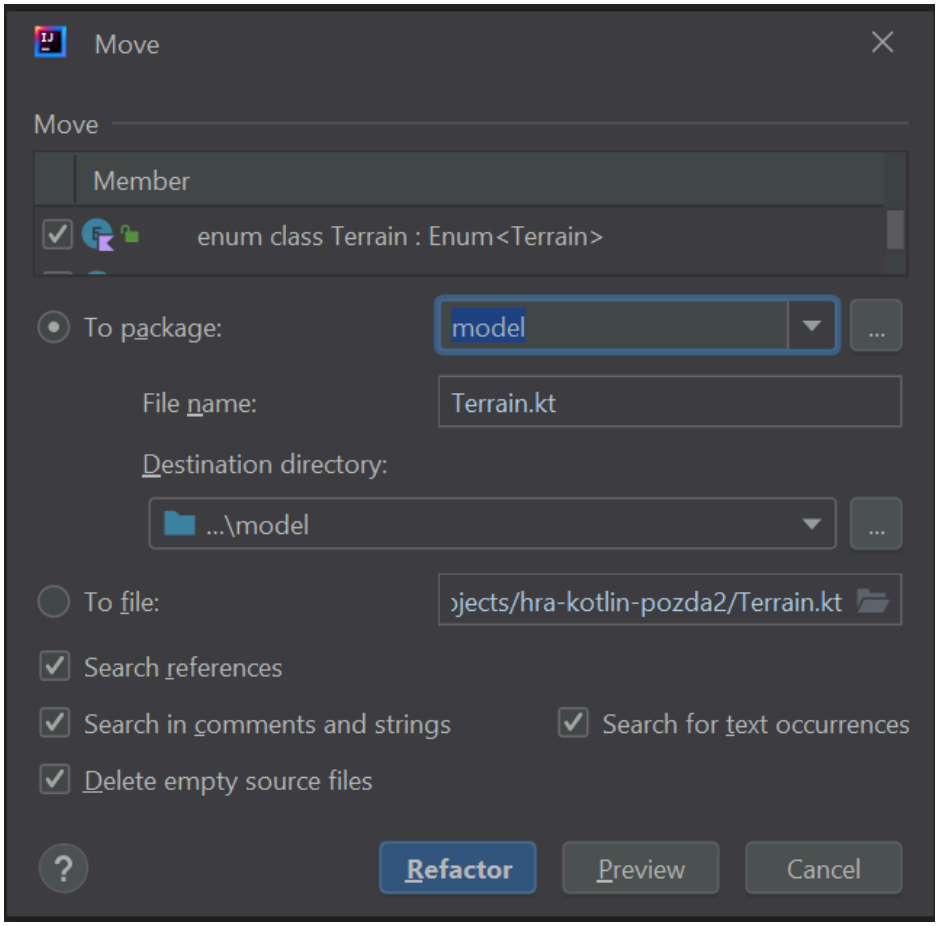




Do package controller přesunume Main.



Zbylé zdrojové soubory přesuneme do package model.



Hrdina

Ve package model vytvoříme data class Hero.

Hrdina bude mít vlastnosti name, position, health, attack, defense, healing. Rovnou nastavíme mimo pozice výchozí hodnoty, abychom je nemuseli hned zadávat.

```
data class Hero (var name: String = "Hrdina",
                 var position: Position,
                 var health: Double = 100.0,
                 var attack: Double = 1.0,
                 var defense: Double = 1.0,
                 var healing: Double = 0.5){

}
```

Protože budeme chtít informovat hráče o stavu hrdiny v přehledné formě, přetížíme metodu toString.

Stejně jako v javě můžeme slučovat Stringy pomocí +, ale nejedná se o efektivní operaci. Proto použijeme StringBuilder.

V kotlinu můžeme v rámci řetězce vypsat hodnotu pomocí $promenna jak je vidět při výpisu jména. Nebo můžeme používat operátor + a výstup proměnné formátovat.

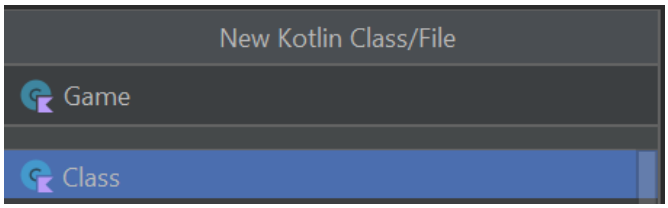
Metoda toString musí vracet String, proto ho nesmíme zapomenout pomocí return vrátit.

```
override fun toString(): String {
    val description = StringBuilder("")
    description.append("Stav hrdiny\n")
    description.append("===========\n")
    description.append("Jméno        $name \n")
    description.append("Zdraví:      "+ "%.2f".format(health) + "\n")
    description.append("Útok:        "+ "%.2f".format(attack) + "\n")
    description.append("Obrana:      "+ "%.2f".format(defense) + "\n")
    description.append("Uzdravování: "+ "%.2f".format(healing) + "\n")
    return description.toString()
}
```

Hra

V package controller máme soubor Main, který je zatím generuje herní plán a ukazuje mapu.

Vytvoříme si třídu Game v package Controller, která bude zastřešovat všechny objekty hry a bude obsahovat metody pro řízení běhu hry.




```
class Game () {

}
```

Třída bude obsahovat základní parametry hry jako rozměr herní plánu a počet lesů.

```
private var width = 20
private var height = 10
private var numForests = 4
```

Přidáme instanci herního plánu, který rovnou vygenerujeme.

```
private var gamePlan = GamePlan (width, height, numForests)
```

Vytvoříme i hrdinu. Protože všechny vlastnosti až na pozici mají výchozí hodnoty, stačí při spouštění konstruktoru získat z gamePlan náhodnou pozici na louce. Kdo neudělal úkol, musí si doplnit i generateRandomPositionOnMeadow()

```
private var hero = Hero (position = gamePlan.generateRandomPositionOnMeadow()
```

Budeme potřebovat i proměnnou pro načtení vstupu od uživatele.

```
private var command : String = ""
```

Vytvoříme hlavní metodu run(), která bude obsahovat hlavní cyklus hry.

```
fun run() {
    do {
        command = readCommand()

        if (command.uppercase()=="KONEC") {
            println ("Konec hry")
            break
        }
    } while (true)
}
```

Musíme doplnit funkci readCommand()

```
fun readCommand(): String {
    print ("Zadej příkaz: ")
    return readLine().toString()
}
```

Teď můžeme předělat Main a hru si spustit. I když jediným příkazem, který můžeme zadat je konec.

```
val game = Game()
game.run()
```

Směry

Připravíme si enum class v package model, ve které budou směry, kterými hrdina může táhnout. Základní směry budou NORTH, EAST, SOUTH a WEST.

Hodnoty budou mít i atributy. relativeX a relativeY určují relativní změnu pozice. Například pokud půjde hrdina na sever, position.x se musí zmenšit o 1.

Dalšími atributy hodnoty je překlad do češtiny a příkaz, který je se směrem spojený.

```
enum class Direction (val relativeY : Int, val relativeX : Int, val description: String, val command: String) {
    NORTH (-1, 0, "sever", "s"),
    SOUTH (1, 0, "jih", "j"),
    EAST (0, 1, "východ", "v"),
    WEST (0, -1, "západ", "z")
}
```

Pozice

Připravíme si třídu Position, aby dokázala vytvořit novou pozici ze stávající pozice a směru.

Vytvoříme si sekundární konstruktor, kterému předáme pozici a směr. Ten nejprve musí zavolat primární kostruktor a pak v bloku může spouštět další příkazy.

```
constructor(position: Position, direction: Direction) : this(position.x, position.y) {
    this.x += direction.relativeX
    this.y += direction.relativeY
}
```

Možné příkazy

Hráč bude z klávesnice zadávat příkazy. Hra bude rozpoznávat pouze určité příkazy a ve všech situacích budou všechny příkazy dostupné. Například pokud hrdina stojí u hranice herního plánu nemůže udělat pohyb, aby se dostal na hranici.

Z tohoto důvodu na začátku každého tahu vytvoříme seznam možných příkazů. Možné příkazy si budeme ukládat do ArrayList<String>. Ten nám proti běžnému poli umožní seznam čistit a přidávat možné příkazy.

```
private var possibleCommands = arrayListOf<String>()
```

Do hlavního cyklu - metoda run, přidáme na začátek pročištění seznamu a zavolání metody, která dokáže zjistit, jaké jsou možné příkazy.

```
fun run() {
    do {
        possibleCommands.clear()
        setPossibleCommands(hero)
```

Teď musíme implementovat metodu setPossibleCommands, které předáme jako vstupní parametr objekt hrdiny, ze které dokáže získat jeho aktuální polohu. Jsou tři základní příkazy, které hráč může dát kdykoliv.

```
fun setPossibleCommands(hero: Hero) {
        possibleCommands.add("stav")
        possibleCommands.add("mapa")
        possibleCommands.add("konec")
    }
```

Pro každý podporovaný směr musím vytvořit Position z pozice hrdiny a relativních souřadnic směru. K této pozici v herním plánu zjistím herní pole a z něj si přečtu, jestli je jeho terén schůdný.

```
if (gamePlan.getGameField(
        Position (hero.position.x + Direction.NORTH.relativeX,
            hero.position.y + Direction.NORTH.relativeY))
        .isWalkable()) possibleCommands.add(Direction.NORTH.command)
if (gamePlan.getGameField(
        Position (hero.position.x + Direction.SOUTH.relativeX,
            hero.position.y + Direction.SOUTH.relativeY))
        .isWalkable()) possibleCommands.add(Direction.SOUTH.command)
if (gamePlan.getGameField(
        Position (hero.position.x + Direction.EAST.relativeX,
            hero.position.y + Direction.EAST.relativeY))
        .isWalkable()) possibleCommands.add(Direction.EAST.command)
if (gamePlan.getGameField(
        Position (hero.position.x + Direction.WEST.relativeX,
            hero.position.y + Direction.WEST.relativeY))
        .isWalkable())  possibleCommands.add(Direction.WEST.command)
```

Kód lze vylepšit, že využijeme schopnosti enum třídy vracet hodnoty pomocí Direction.values() nebo `enumValues<Direction>().forEach { println(it.command) }`

Hra po provedení tahu komunikovat s uživatelem a říci mu, v jaké se nachází situaci. Vytvoříme metodu, kdy se hrdina rozhlédne po okolí a vrátí řetězec s popisem co vidí.

Opět k tomu použijime StringBuilder a podobnou funkcionalitu jako u zjišťování prostupnosti okolních políček. Tedy vytvoříme se Position z pozice hrdiny a relativních souřadnici směru. Získáme herní pole, podíváme se na terén a jeho popis.

```
fun getSurroundingDescription () : String {
    val description = java.lang.StringBuilder("")
    description.append ("Na severu je "+ gamePlan.getGameField(
        Position(hero.position, Direction.NORTH)).terrain.description+". ")
    description.append ("Na východu je "+ gamePlan.getGameField(
        Position(hero.position, Direction.EAST)).terrain.description+". ")
    description.append ("Na jihu je "+ gamePlan.getGameField(
        Position(hero.position, Direction.SOUTH)).terrain.description+". ")
    description.append ("Na západu je "+ gamePlan.getGameField(
        Position(hero.position, Direction.WEST)).terrain.description+".")

    return description.toString()
}
```

Nyni rozšíříme hlavní cyklus.

```
fun run() {
    do {
        possibleCommands.clear()
        setPossibleCommands(hero)
        println ("-----------------------------------------------")
        println (getSurroundingDescription())
        println ("Možné příkazy jsou: $possibleCommands")
        command = readCommand()

        if (command.uppercase()=="KONEC") {
            println ("Konec hry")
            break
        }
    } while (true)
}
```

Teď zbývá reagovat na příkazy, které zadal hráč. Vytvoříme si metodu runCommand. Protože některé příkazy mohou vracet zpětnou vazbu hráči, bude metoda vracet String, který vypíšeme na obrazovku.

Pro příkaz stav vrátíme hero.toString. Pro příkaz mapa zavoláme metodu gamePlan.map(). Aby metoda byla kompatibilní s ostatními, neměla by přímo vypisovat na obrazovku, ale měla by vrátit řetězec. To můžete upravit.

```
fun runCommand(command: String) : String {
    when (command) {
        "stav" -> return (hero.toString())
        "mapa" -> gamePlan.map()
    }
    return ""
}
```

Hrdina se musí pohybovat. Musíme mu přidat metodu go, která změnu pozice provede. Vstupním parametrem bude směr Direction.

```
fun go (direction: Direction) : String {
    position.x += direction.relativeX
    position.y += direction.relativeY
    return "Jdu na ${direction.description}"
}
```

Předposledním krokem je doplnění when ve funkci runCommand o směry.

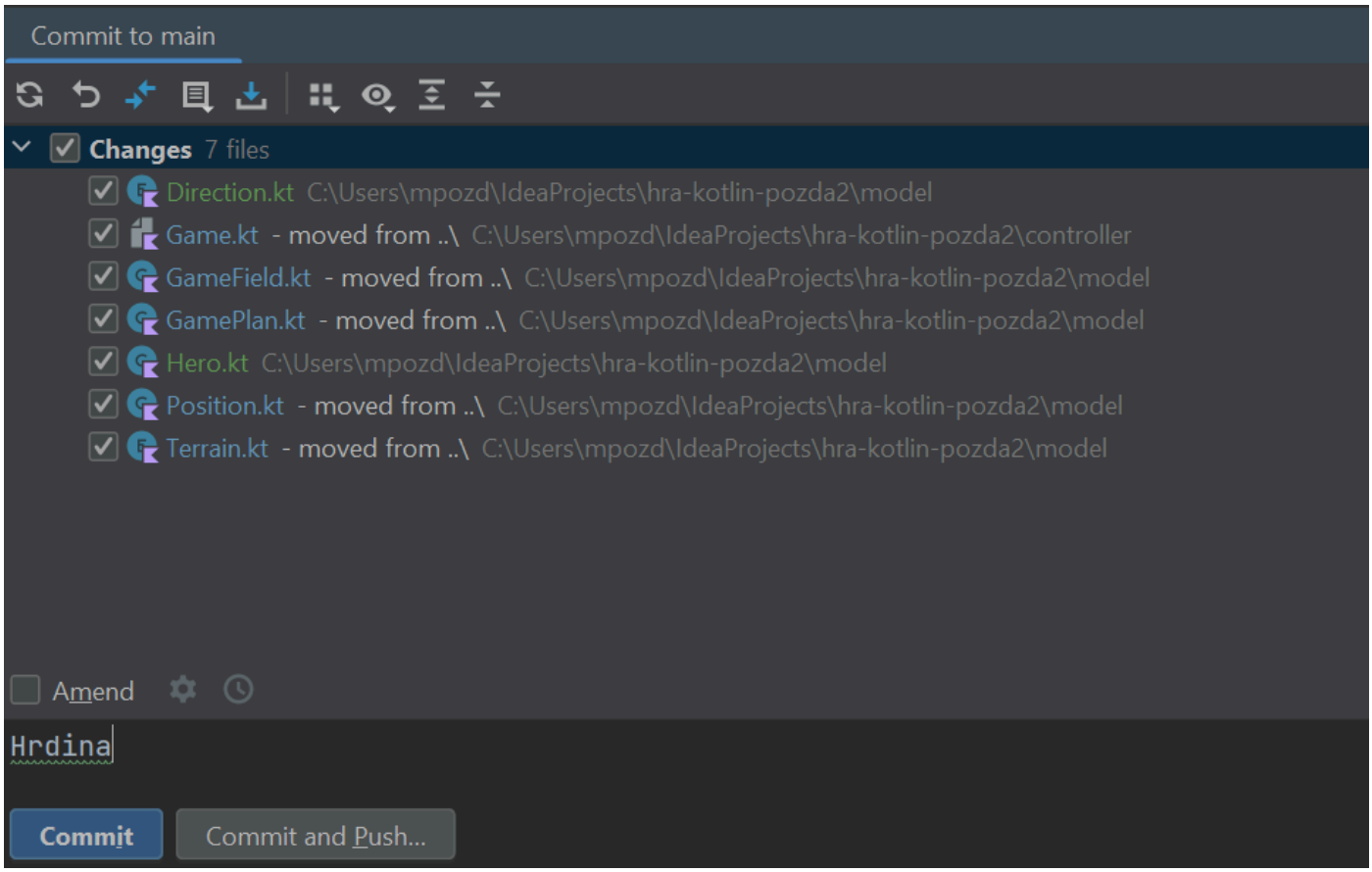
```
fun runCommand(command: String) : String {
    when (command) {
        "stav" -> return (hero.toString())
        "mapa" -> gamePlan.map()
        Direction.NORTH.command -> return hero.go(Direction.NORTH)
        Direction.EAST.command -> return hero.go(Direction.EAST)
        Direction.SOUTH.command -> return hero.go(Direction.SOUTH)
        Direction.WEST.command -> return hero.go(Direction.WEST)
    }
    return ""
}
```

Posledním krokem je zavolání funkce runCommand()

```
command = readCommand()
println(runCommand(command))
```

Git

Pokud vše funguje, nezapomeňte změny potvrdit a pushnout na git.



Naposledy změněno: čtvrtek, 2. května 2024, 15.42

◄ Kotlin - arrayList, cykly - video

Přejít na...

Cvičení - úkol (kontrola vstupu, více směrů, mapa) ►

Nápověda a dokumentace (anglicky)
Kontaktujte podporu stránek

Jste přihlášeni jako Lukáš Čegan (Odhlásit se)
DANTE_MA_FINAL

Moje stránka
Kalendář
Kontakty
Mahara
Kurzy
Čeština (cs)

Čeština (cs)
English (en)

Souhrn uchovávaných dat

Stáhněte si mobilní aplikaci

Mobilní aplikace

Kotlin - dědičnost - text

Nepřátelé

V tomto cvičení přidáme do hry nepřátele. Při vytváření nepřítele budeme mít několik omezení

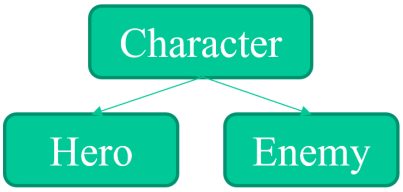
- Nepřítel stojí na volném poli, které má terén louka. Postavu tedy neumístíme do řeky, lesa. Nebudou dva nepřátelé na jednom políčku.
- Na začátku nebude nepřítel ani na stejném herním poli jako je hrdina.
- Nepřítel se během hry nebude pohybovat, bude pouze čekat až na jeho herní pole vstoupí hrdina.
- Nepřítel bude mít vlastnosti jako je jméno, zdraví, síla útoku a síla obrany.
- Ve hře bude existovat více typů nepřátel, kteří se budou lišit vlastnostmi.

Dědičnost

Když se podíváme na hrdina a nepřítele, tak na první pohled mají některé stejné vlastnosti (jméno, pozice, zdraví, útok nebo obrana). Stejně tak budou vykonávat podobné činnosti - obrana a útok.

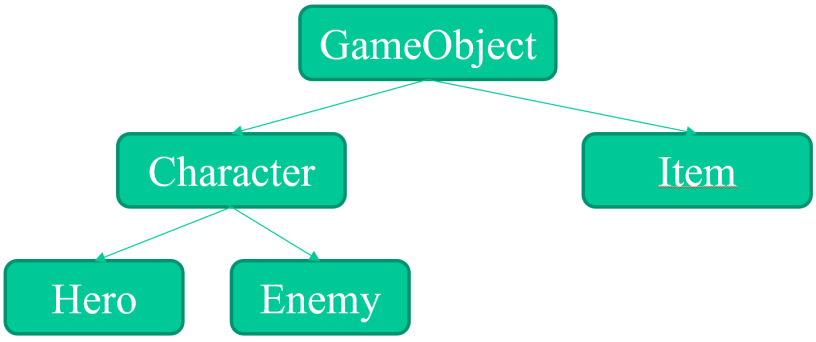
Mohli bychom je implementovat nezávisle, ale některý kód bychom duplikovali, proto použijeme dědičnost.

Vytvoříme si třídu Character. Její potomci budou Hero a Enemy.



V budoucnu budeme rozšiřovat hru i o předměty. Předměty hrdinovi po sebrání zlepšují jeho statistiky. Předmět a Postava mají opět nějaké společné vlastnosti jako jméno a pozici. Proto naši dědičnost rozšíříme o třídy GameObject. Potomci této třídy budou Character a později Item.

Konečné vztahy tříd ve hře budou vypadat následovně:



Herní objekt

Vytvoříme si v package model novou abstraktní třídu GameObject.

Třída bude abstraktní, tzn. že nemusí mít implementované metody. Atributy musí být open, aby je potomci mohli měnit.

lateinit

Kotlin umožňuje deklaraci proměnných tak, že rovnou instanci přiřazujeme hodnotu nebo null hodnotu.

```
var variable : CustomClass = CustomClass()
var variable : CustomClass? = null
```

V některých případech, ale nechceme, aby proměnná mohla obsahovat null hodnotu. Při volání metod bychom museli používat variable?.someMethodCall() nebo variable!!.someMethodCall()

Ale zároveň v místě deklarace ještě neznáme její hodnotu. lateinit říká, že proměnná nesmí mít null hodnotu a konkrétní hodnotu promměné přiřadíme později.

Případně můžeme použít abstract.

```
abstract class GameObject {
    open lateinit var name : String
    abstract var position: Position
}
```

Postava

Vytvoříme abstraktní třídu Character, která budě dědit atributy name a position ze třídy Character. Do třídy přidáme další 3 atritbuty health, attack a defense, který určíme nějakou počáteční hodnotu, abychom je nemuseli definovat jako lateinit

```
abstract class Character : GameObject() {
    open var health: Double = 100.0
    open var attack: Double = 1.2
    open var defense: Double = 1.0
}
```

Hrdina a nepřítel mají stejné chování. Pokud jejich zdraví poklesne k nule, jsou mrtví. Do třídy character implementujeme tedy metodu isDead.

```
fun isDead (): Boolean {
    return (health < 0.00001)
}
```

Nepřítel

Vytvoříme třídu Enemy, která bude dědit z Character. Opět ji vytvoříme jako data class, protože se nám na pozadí automaticky vygenerují metody equals(), toString, copy(). Data class nemohou být abstraktní a vyžadují minimálně jeden parametr v konstruktoru.

Protože ve třídě GameObject jsme použili abstrakt, musí se name a position v konstruktoru objevit. Používá se před nimi override, kterým dáváme najevo, že přetěžujeme jejich implementaci z abstraktní třídy. Parametry healt, attack a defense uvádět nemusíme, protože ty mají výchozí hodnotu.

```
data class Enemy(override var name: String, override var position: Position) : Character() {}
```

Pro budoucí použití, kdy budeme vytvářet různé typy nepřátel by se nám hodil, ale konstruktor, kterému můžeme předat všechny parametry. Proto použijeme raději tuto definici.

```
data class Enemy(
    override var name: String = "",
    override var position: Position,
    override var health: Double = 100.0,
    override var attack: Double = 1.0,
    override var defense: Double = 1.0) :
    Character ()
```

Hrdina

Hrdina bude dědit ze třídy Character. Proto upravíme jeho implementaci.

Atributy health, attack, defense jsou zděděny z Character, tak je uvádět nemusíme. Konstruktor bude pouze potřebovat přetížení abstraktních atributů name a position.

Proti třídě Character, má hrdina atribut healing, který určuje okolik se hrdina automaticky uzdraví za kolo.

```
data class Hero (override var name: String = "Hrdina",
                 override var position: Position ) : Character() {
    var healing: Double = 0.5
```

Přidáme i sekundární konstruktor, který musí zavolat primární kostruktor pomocí this (name, position) a pak může následovat další oblok kódu. Sekundární konstruktor časem použijeme, pokud budeme chtít vytvořit hrdinu, který bude mít jiné vlastnosti než výchozí.

```
constructor(name: String, position: Position, health: Double, attack : Double, defense: Double, healing : Double) : this (name,
position) {
    this.health = health
    this.attack = attack
    this.defense = defense
    this.healing = healing
}
```

Hra

Přidáme nepřítele do hry.

Na začátku si vytvoříme proměnnou numEnemies, která bude určovat kolik nepřátel máme vytvořit.

```
private var numEnemies = 5
```

Dále budeme potřebovat seznam všech herních objektů a pro rychlejší práci seznam nepřátel.

```
var enemies = arrayListOf<GameObject>()
var gameObjects = arrayListOf<GameObject>()
```

Vytvoříme blok init, ve kterém budeme postupně generovat herní objekty. Hrdinu máme vygenerovaného z jeho deklarace. Proto ho přidáme do seznamu herních objektů. V seznamu jsou uloženy pouze ukazatelé na instanci objektu. gameObjects.add(hero) žádnou další instanci hrdiny nevytváří.

```
init {
    gameObjects.add(hero)
    generateEnemies()
}
```

Funkce generateEnemies() může vypadat například takto. Několikrát se zavolá funkce generateEnemy. Vysledný objekt se vloží do seznamu herních objektů a seznamu nepřátel.

```
private fun generateEnemies() {
    var enemy : Enemy
    repeat (numEnemies) {
        enemy = generateEnemy()
        enemies.add(enemy)
        gameObjects.add(enemy)
    }
}
```

Samotná funkce generateEnemy vrátí instanci nepřítele. Zavolá se konstruktur třídy, kterému se předají konkrétní parametry a pro vytvoření pozice se zavolá metoda gamePlan.generateFreeRandomPositionOnMeadow(), které je předán seznam doposud vygenerovných herních objektů.

```
private fun generateEnemy(): Enemy {
    return Enemy(name="Skeleton",
        position = gamePlan.generateFreeRandomPositionOnMeadow(gameObjects),
        health = 10.0,
        attack = 5.0,
        defense = 0.5)
}
```

Pozice

Nepřátele se budou generovat na volných pozicích. Proto potřebujeme do třídí Position vložit metodu isFree s parametrem gameObjects. Vstupem do této metody je seznam všech herních objektů. Metoda určí, zda na dané pozici se nevyskytuje nějaký herní objekt.

```
fun isFree (gameObjects: ArrayList<GameObject>): Boolean {
    for (gameObject in gameObjects) {
        if (gameObject.position == this) {
            return false
        }
    }
    return true
}
```

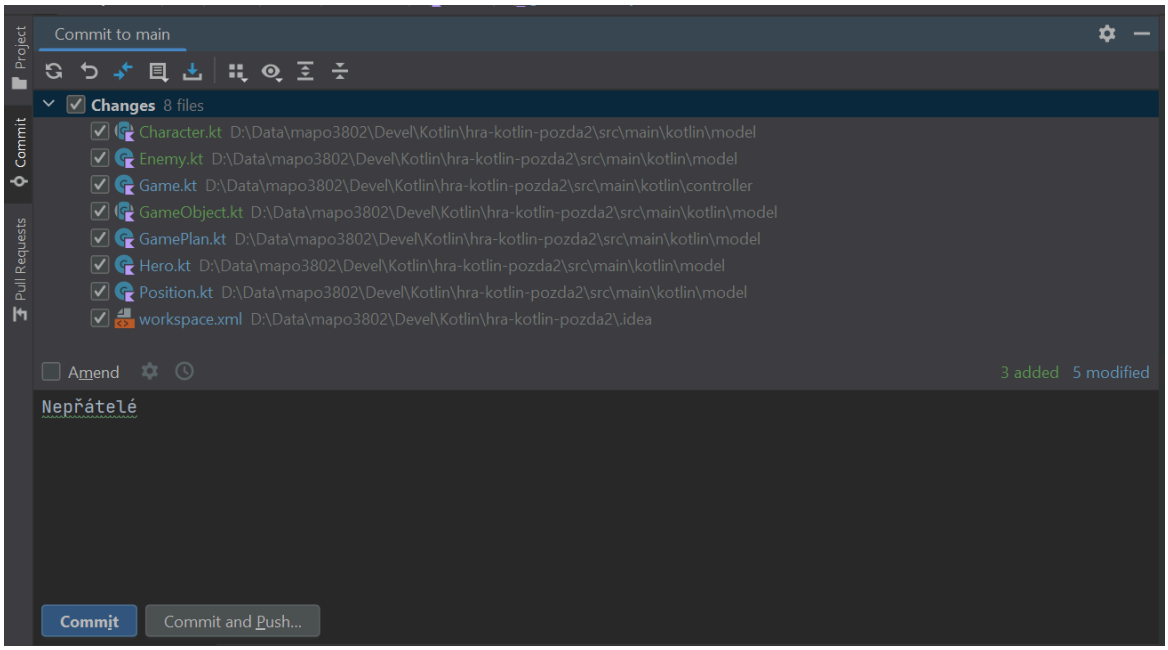
Herní plán

Herní plán obsahuje metodu generateRandomPositionOnMeadow, která vrací náhodnou pozici na louce. Pro generování nepřátel budeme potřebovat metodu generateFreeRandomPositionOnMeadow, který využije metody generateRandomPositionOnMeadow() a isFree()

```
fun generateFreeRandomPositionOnMeadow(gameObjects: ArrayList<GameObject>) : Position {
    var randomPosition : Position
    do {
        randomPosition = generateRandomPositionOnMeadow()
    } while (! randomPosition.isFree(gameObjects) )
    return randomPosition
}
```

Git

Pokud nám vše funguje, změny potvrdíme a pushneme do Repository.



Naposledy změněno: čtvrtek, 2. května 2024, 15.40

◄ Kotlin - dědičnost - video

Přejít na...

Cvičení - úkol (uzdravování, jméno hrdiny, mapa, různí nepřátelé) ►

Mobilní aplikace

Kotlin - Interakce mezi objekty - text

Souboje

V tomto cvičení dokončíme základní implementaci hry. Doprogramujeme souboje a hru upravíme, aby počítala skóre, počty zabití a ukončila se i jinak než napsáním příkazu konec.

Pokud hráč vstoupí na herní pole, kde je živý nepřítel:

- bude o tom informován na obrazovce
- seznam možných příkazů se rozšíří o příkaz utok
- hráč bude moci zaútočit na nepřítele
- v tom samém tahu bez ohledu na příkaz hráče nepřítel provede útok na hráče

Hra bude ukončena

- uživatel napsal příkaz konec
- hrdina je mrtvý
- všichni nepřátelé jsou mrtví

Souboj

Matematický vzorec útoku uděláme jednoduchý. Útočník útočí určitou silou podle své vlastnosti attack. Bránící se má schopnost vykrýt část tohoto útoku svoji vlastností defense.

Výsledný útok je dán rozdílem těchto dvou hodnot. Pokud je obrana větší než útok, tak skutečný útok nevyjde a bude nulový.

Případný pozitivní skutečný útok se odečte od zdraví bránicího se.

Při souboji interagují dva objekty a posílají si zprávy/volají svoje metody getter a setter.

Výsledek souboje pak vrátíme jako řetězec, aby měl hráč zpětnou vazbu.

```
open fun attack (enemy: Character) : String {
    var realAttack = attack - enemy.defense
    if (realAttack < 0) realAttack=0.0
    enemy.health -= realAttack

    if (enemy.isDead()) return "${enemy.name} je mrtvý."
    return "$name zaútočil silou " + "%.2f".format(realAttack) +"."
}
```

Hrdina

V rámci hrdiny budeme evidovat počet zabití. Do třídy Hero vložíme proměnnou kill

```
var kills: Int = 0
```

Upravíme metodu toString, aby počet zabití vypisovala.

```
description.append("Uzdravování: "+ "%.2f".format(healing) + "\n")
description.append("Zabití:      $kills \n")
return description.toString()
```

U hrdiny přetížíme metpdu attack zděděnou ze třídy Character, aby aktualizovala počet zabití. Přetížení se označujeme pomocí override.

Protože souboj zůstává zachování, můžeme zavolat pomocí super, rodičovskou implementaci metody. Pak jen doplníme kód na aktualizaci statistiky.

```
override fun attack(enemy: Character): String {
    val result = super.attack(enemy)
    if (enemy.isDead()) kills += 1
    return result
}
```

Hra

Provádění souboje musíme doplnit na třídy Game. Postupně budeme rozšiřovat různé metody.

Skóre hry budeme určovat počtem tahů potřebným pro dokončení hry. Založíme se ve třídě Game proměnnou score.

```
var score = 0
```

Dále se vytvoříme metodu getEnemyOnGameField, která nám vrátí odkaz na nepřítele na zadané pozici.

```
fun getEnemyOnGameField (position: Position, gameObjects: ArrayList<GameObject>) : Enemy? {
    for (obj in gameObjects ) {
        if (obj is Enemy) {
            if (obj.position == position) {
                return obj
            }
        }
    }
    return null
}
```

Hráči musíme povolit příkaz pro útok, pokud se nachází na stejném herním poli jako nepřítel. Na začátek metody setPossibleCommands() přidáme zavolní připravené metody. Null hodnotu můžeme rovnou otestovat dotaz na třídu objektu.

```
val enemy = getEnemyOnGameField(hero.position, enemies)
if (enemy is Enemy && ! enemy!!.isDead()) possibleCommands.add("utok")
```

Stejně tak musíme uživateli dát zprávu, pokud se na pozici hrdiny nachází nepřítel. Do metody getSurroundingDescription() přidáme následující řádky.

```
val enemy = getEnemyOnGameField(hero.position, enemies)
if (enemy != null && ! enemy!!.isDead()) description.append("\nPozor ${enemy!!.name}.")
if (enemy != null && enemy!!.isDead()) description.append("\nNa zemi vidíš mrtvolu ${enemy!!.name}.")
```

Do metody runCommand přidáváme na začátek zvýšení počtu tahů a upravíme when, aby umožňoval provést příkaz utok.

```
score++
val enemy = getEnemyOnGameField(hero.position, enemies)



            "utok" -> return if (enemy != null) {
                hero.attack(enemy)
            } else {
                ""
            }
```

Vytvoříme metodu enemyAttack(), která provede zpětný útok nepřítele na hrdinu. Pokud v proměnné enemy zjištěné na začátku cyklu máme objekt třídy Enemy, můžeme otestovat, že nepřítel není mrtvý. Pokud je živý, nepřítel zaútočí na hrdinu a výsledek souboje vrátíme.

```
fun enemyAttack() : String {
    val enemy = getEnemyOnGameField(hero.position, enemies)

    if (enemy is Enemy) {
        if (!enemy.isDead()) {
            return(enemy.attack(hero))
        }
    }
    return ""
}
```

Pro otestování konce hry implementujeme metodu allEnemiesDead() a isGameFinish(), která otestuje různé možné konce hry a případně vrátí zprávu. Pokud hra nemá končit vrátí prázdný řetězec.

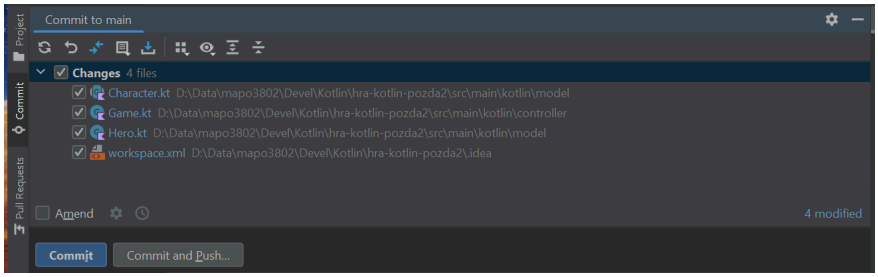
```
fun allEnemiesDead() : Boolean {
    for (enemy in enemies) {
        if (enemy is Enemy && ! enemy.isDead()) return false
    }
    return true
}

fun isGameFinished(): String {
        if (hero.isDead()) return "Jsi mrtvý."
        if (allEnemiesDead()) return "Všichni nepřátelé jsou mrtví. Vyhrál jsi. Potřeboval jsi $score tahů."
        if (command == "konec") return "Konec hry."
        return ""
    }
```

Nyní nám už chybí jen upravit hlavní cyklus. Po provedení příkazu hráče, spustíme útok nepřítele, zvýšíme skóre a otestujeme různé podmínky pro ukončení hry.

```
fun run() {
   var message = ""
   do {
            possibleCommands.clear()
            setPossibleCommands(hero)
            println ("-----------------------------------------------")
            println (getSurroundingDescription())
            println ("Možné příkazy jsou: $possibleCommands")
            command = readCommand()
            println (runCommand(command))
            println (enemyAttack())

            message = isGameFinished()
            if (message.isNotEmpty()) {
                println (message)
                break
            }
    } while (true)
}
```

Git

Nezapomene potvrdit změny a uložit je na vzdáleném repository.



Naposledy změněno: čtvrtek, 2. května 2024, 15.38

◄ Kotlin - Interakce mezi objekty - video

Přejít na...

Cvičení - úkol (kácení lesa, předměty) ►

Nápověda a dokumentace (anglicky)
Kontaktujte podporu stránek

Jste přihlášeni jako Lukáš Čegan (Odhlásit se)
DANTE_MA_FINAL

Moje stránka
Kalendář
Kontakty

Mobilní aplikace

Vytvoření UI (úvod do Android Studia) - text

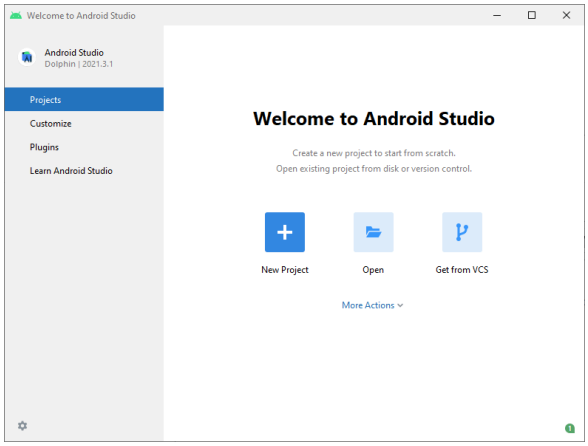
Git

Mobilní aplikaci budeme vytvářet v Android studio. Dosavadní práci ponecháme v původním repostitory a mobilní aplikaci budeme vytvářet v novém Git repository.

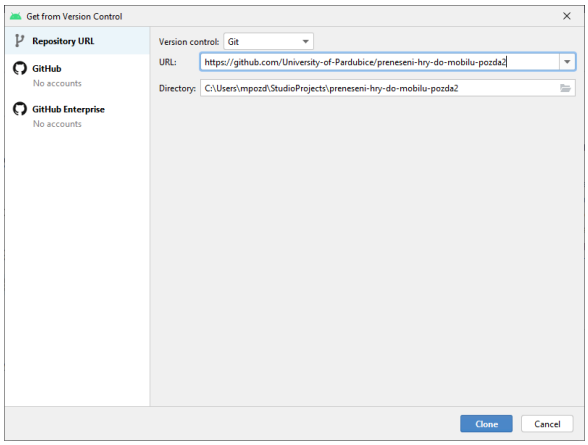
To si založte podobně jako ve cvičení 3 na tomto odkazu - github classroom

Založení projektu

Založte projekt z gitu (Get from VCS)



Vložte URL nového repository



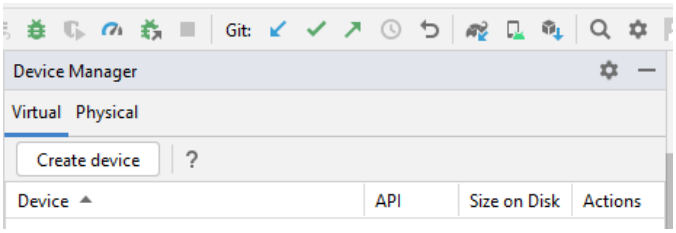
Při klonování budete vyzvání k přihlášení k účtu na gitu. Použijte Token nebo Login via Github (viz cvičení 3).

Po dokončení klonování si Android studio začne stahovat potřebné balíky a začne projekt kompilovat. Může to chvíli trvat.

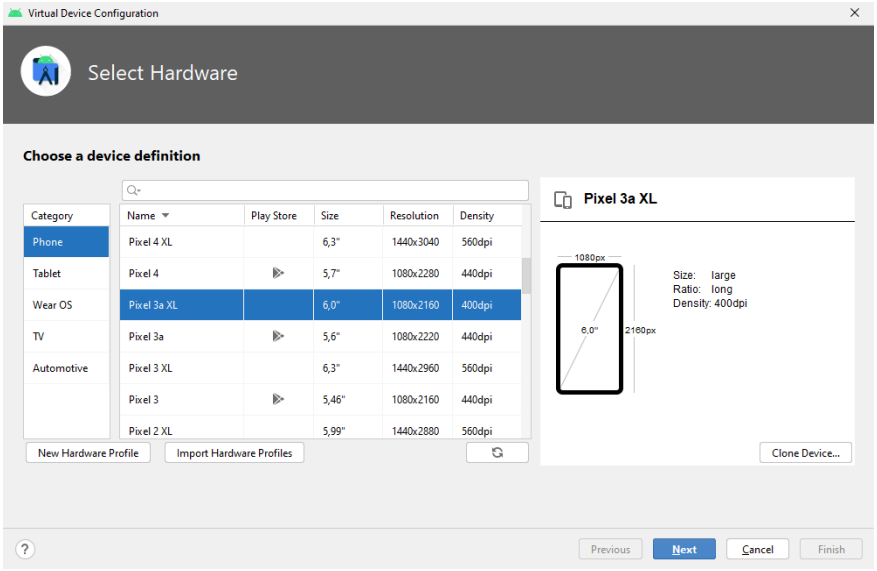
První spuštění

Aplikaci při vývoji můžeme spouštět na připojeném mobilu, který má povolené ladění.

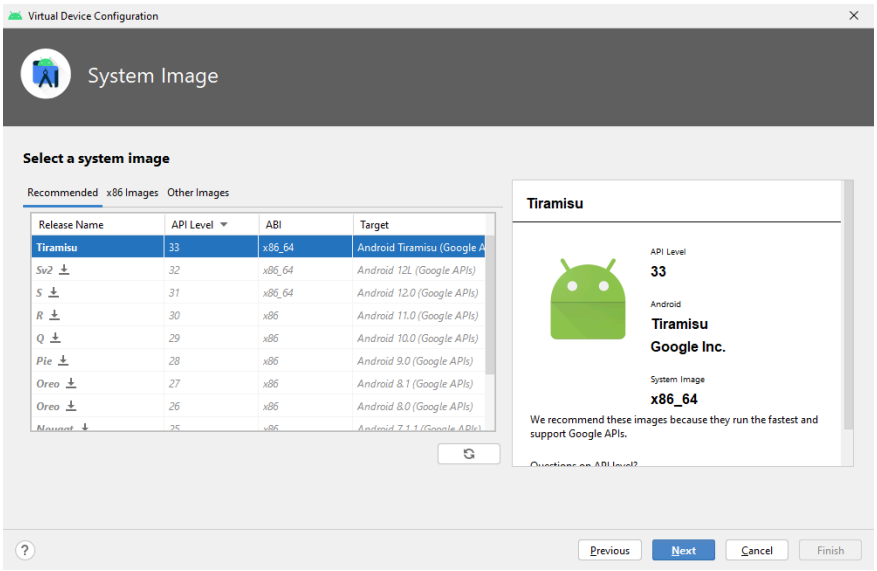
Nebo v Device Manageru vytvoříme Virtuální zařízení

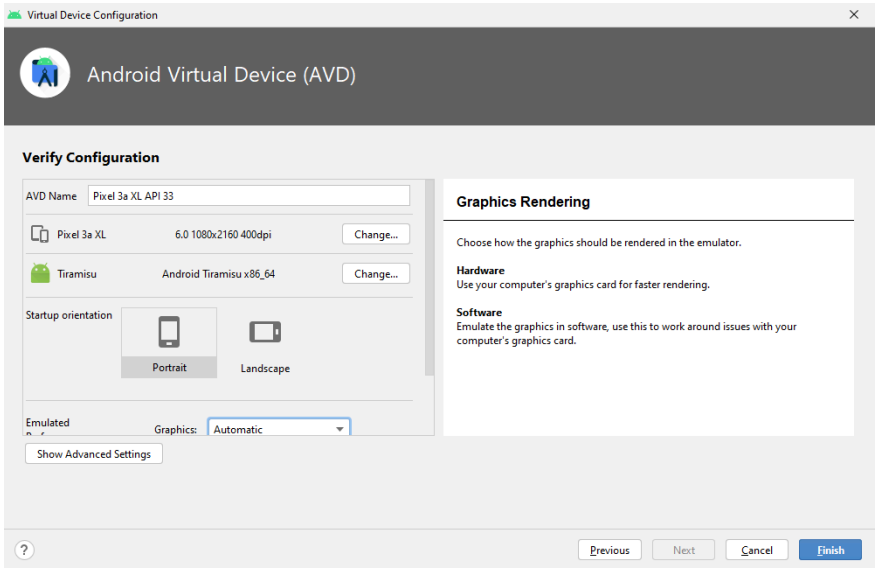


Vybereme si vhodné zařízení, pro které budeme aplikaci vytvářet.

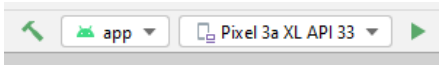


A vybereme mu systémový image. Je možné, že zatím není na počítači používané a bude se stahovat.





Prvotní aplikaci, teď můžeme na Emulátoru spustit.

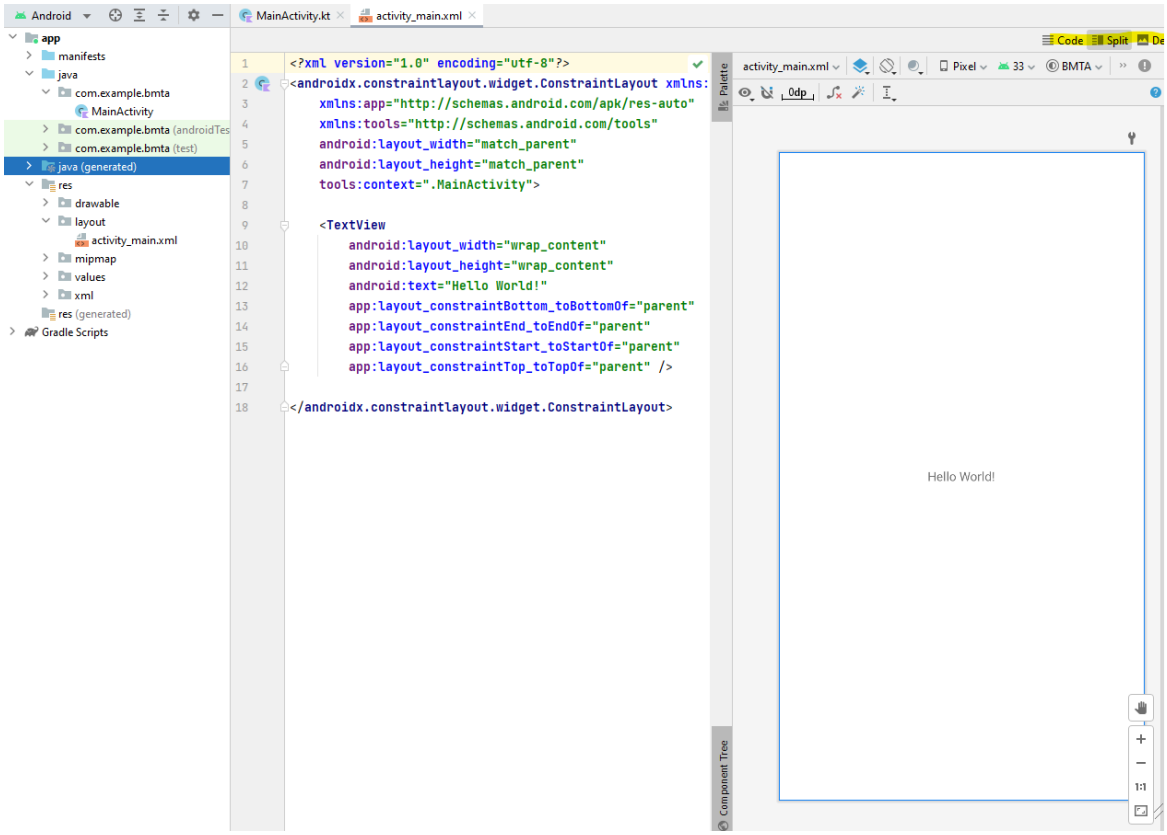


Aktivita

Aktivita je základní jednotka (třída) aplikace pro android. Životní cyklus aktivity a volání dalších aktivity a navigace mezi nimi je základní filosofie, jak funguje aplikace pro android.

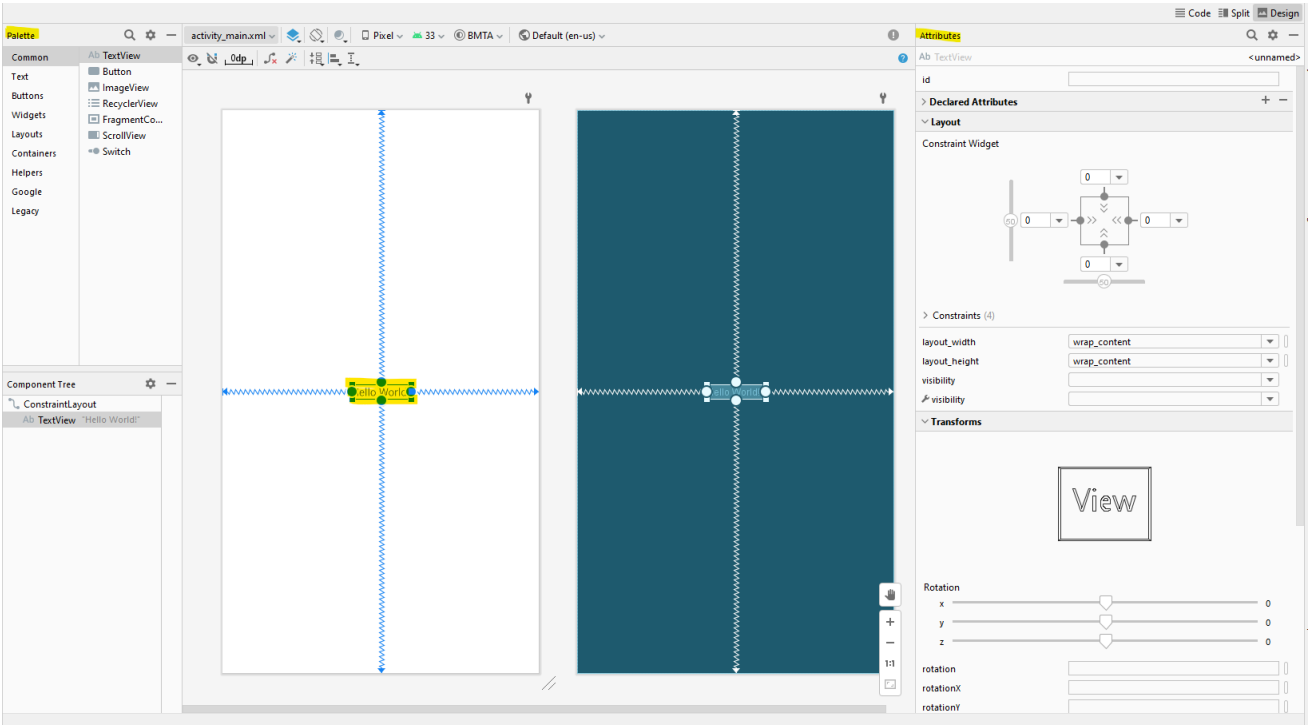
V projektu máme vytvořenou prázdnou aktivitu MainActivity. Skládá se z programu MainActivity.kt ve složce Java a ze souboru activity_main.xml ve složce resources/layout. Zjednodušeně řečeno XML soubor se stará o vzhled aktivity a program o ošetření uživatelských vstupů.

Když otevřeme soubor activity_main.xml, můžeme se přepínat mezi pohledy čistého XML kódu, grafického návrhu a sloučeného pohledu.



Vzhled obrazovky můžeme definovat přímou editací XML kódu nebo použít pohledu Design, kde si vzhled navrhneme interaktivně.

V paletě máme dostupné různé UI prvky, které můžeme přetáhnout na plochu aktivity. Označením prvku se vpravo zobrazí atributy prvku, kterými můžeme měnit vzhled a chování.

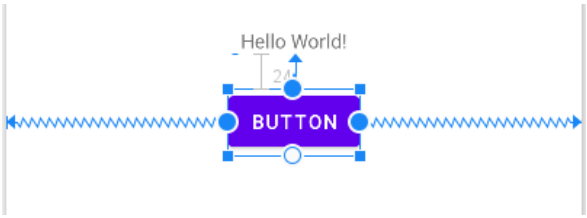


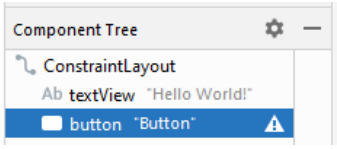
Při návrhu vzhledu je důležitý pojem layout. Layout definuje pravidla, jak se jednotlivé prvky na obrazovce umístí. Jednoduchým je například Linear Layout, který budeme použít u seznamů, kdy jsou jednotlivé prvky automaticky umístěny v řadě za sebou.

Pro návrh komplikovanějšího rozmístění se používá Constraint Layout. Každý UI prvek má 4 body - bottom, top, start (levá strana), end (pravá strana). Tyto body se vážou na jiné objekty. Každý prvek musí mít definovaný alespoň jeden vertikální a jeden horizontální bod.

Na obrázku výše je vidět, že text Hello world, je ukotvený na všechny 4 strany layoutu. Proto je ve středu aktivity.

Do aktivity přidáme tlačítko Button. Start a End navážeme na kraje layoutu a horní bod navážeme na spodní bod Hello world. Tím vzniká řetěz závislotí UI prvků, jak je vidět v Component Tree.



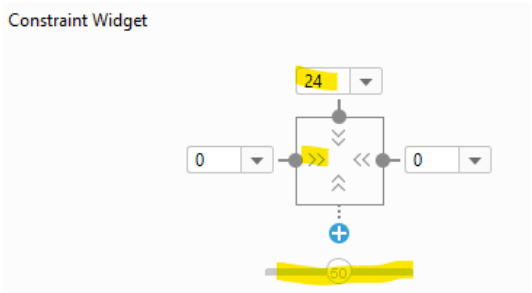


V atributech prvku lze umístění měnit v Constraint Widget. Můžeme přidat například mezeru mezi body (24), kliknutím na >> měníme šířku prvku z dynamické na fixní nebo roztaženou.

Velikost je vhodné uvádět v dp, aby grafický návrh byl relativně nezávislý na zařízení:

- px - skutečný pixel
- dp - device-independent pixels - velikost bodu podle DPI display, základ je 160x160 px=dp, při dpi 320, 1dp = 2px
- pt - bod - 1/72 fyzické velikosti obrazovky v inch
- sp - scalable pixel - jako dp, ale používá se pro fonty

Tlačítko button jsme ukotvili na levou a pravou stranu layoutu. Proto je uprostřed. Jeho relativní pozici můžeme měnit posuvníkem - měníme bias (procentuální umístění středu prvku v možném prostoru)



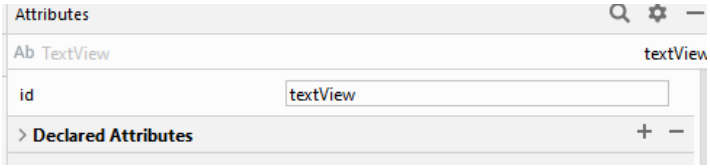
Můžeme spustit program, abychom ověřili, že v návrhu máme všechny podmínky umístění.

Základní interakce XML a programu

Při vytváření aktivity se volá metoda onCreate, která zavolá předka metody a pak propojuje program s XML pomocí setContentView, ve kterém se odkážeme v resource na jméno layoutu.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

Pokud chceme ovlivňovat z programu UI prvek, musíme vědět jeho id, to se zjistí v XML. Je vhodné UI prvky metodicky přejmenovávat a podle určité konvence a tu v dodržovat.



Abychom se mohli v programu na prvek odkazovat musíme si vytvořit v programu proměnnou, kterou pomocí **findViewById** provážeme s UI v layoutu. Pak prvku můžeme například změnit text. Kód vkládáme do metody **onCreate**

```
val text = findViewById<TextView>(R.id.textView)
text.text = "Ahoj světe"
```

Zkusíme přidat reakci, kdy se text změní až po stisku tlačítka. Zadefinujeme si proměnnou pro tlačítko a vytvoříme mu listner - kód který se vyvolá při stisknutí tlačítka. Kód vkládáme do metody onCreate

```
val text = findViewById<TextView>(R.id.textView)
val button = findViewById<Button>(R.id.button)

button.setOnClickListener {
    text.text = "Ahoj světe"
}
```

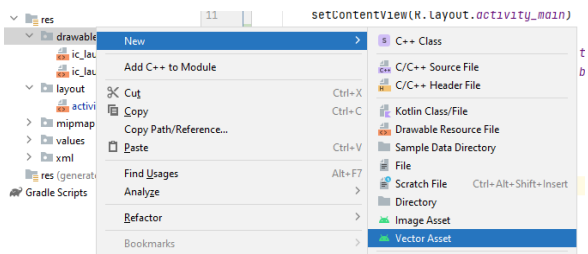
Program můžeme opět vyzkoušet.

Návrh úvodní obrazovky

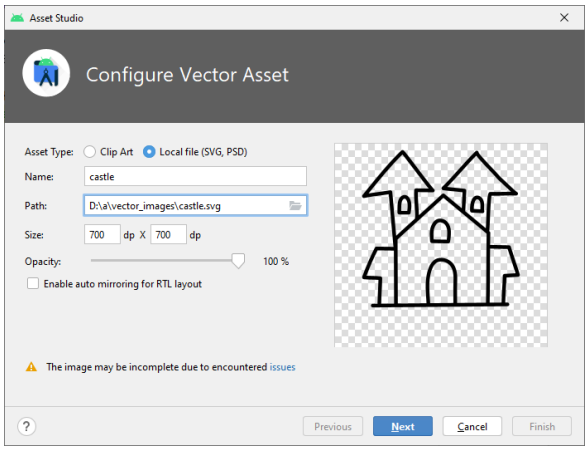
Na úvodní obrazovku přidáme obrázek. Můžeme používat klasické bitmapové obrazovky a velmi často se vektorové obrázky, které se dynamicky přizpůsobují různým obrazovkám bez velkých zkreslení.

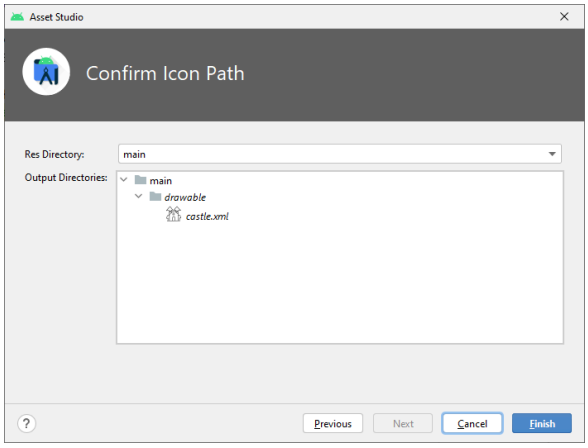
Vektorové obrázky si můžete stáhnout z moodlu. Případně se dají najít na internetu volně dostupné hezčí :-)

Vytvoříme si Vector Asset.

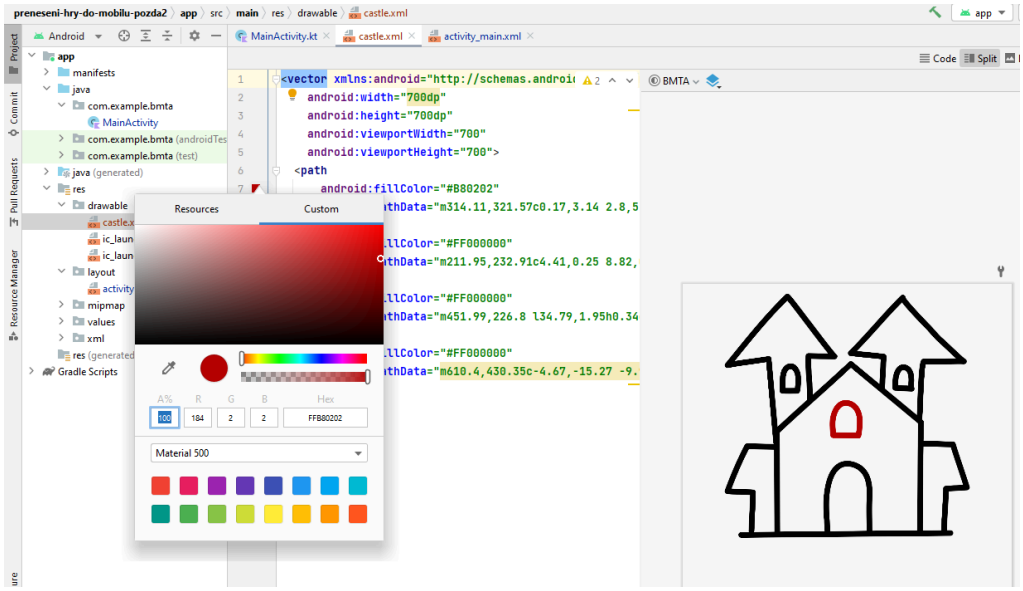


Vytvoříme si obrázek hradu.

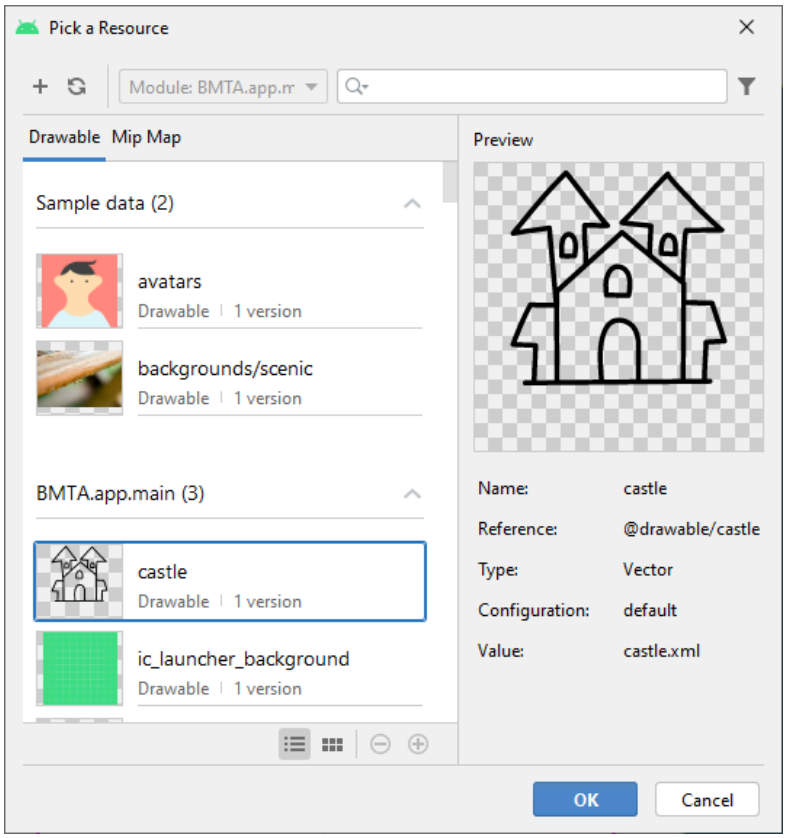




Když si obrázek otevřeme, můžeme například změny barvy jednotlivých cest kliknutím na barvy vedle čísla řádky.



V aktivitě smažeme textView. Přidáme **imageView**, kterému vybere hrad a velikost obrázku přizpůsobíme.

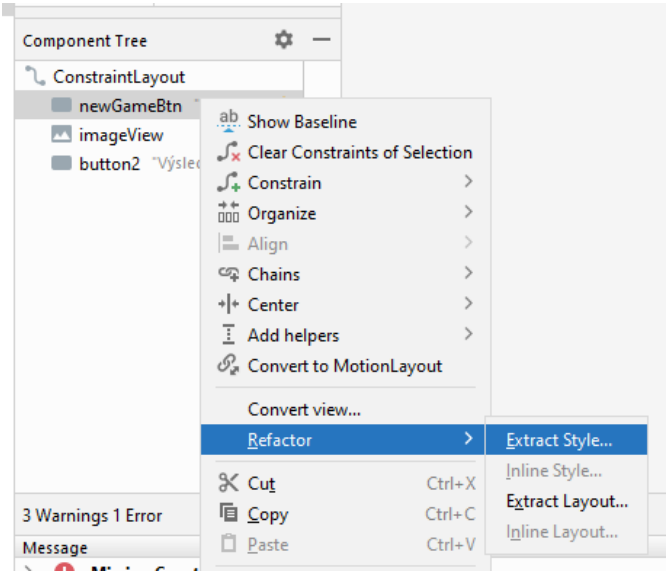


Tlačítko button přejmenujeme na **newGameBtn**. Potvrdíme Refactor.

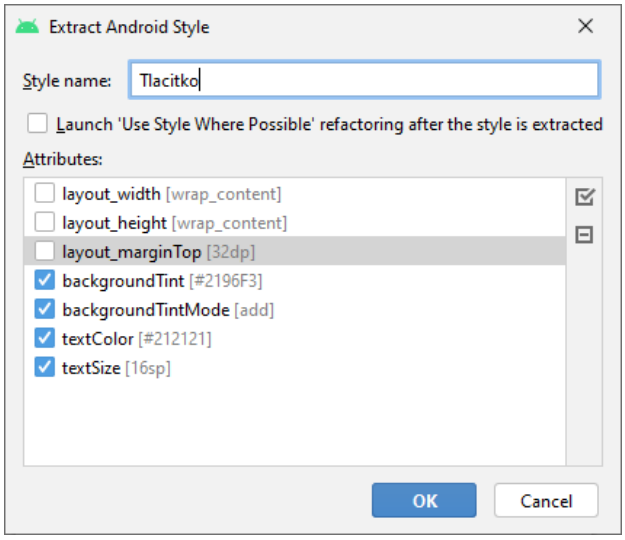
Tlačítku změníme mu text na Nová hra a upravíme barvu pomocí atributu **backgrountTint**. Změníme barvu nápisu pomocí atributu **textColor**, velikost písma pomocí **textSize**.

Přidáme další tlačíko s id **resultsBtn** a nápisem **Výsledky**. Mohli bychom všechny předchozí nastavení vzhledu tlačítka zopakovat, ale je praktičtější si udělat styl.

Označíme si v Component tree nastavené tlačítko a zvolíme v kontextovém menu **Refactor** / **Extract Style** ...



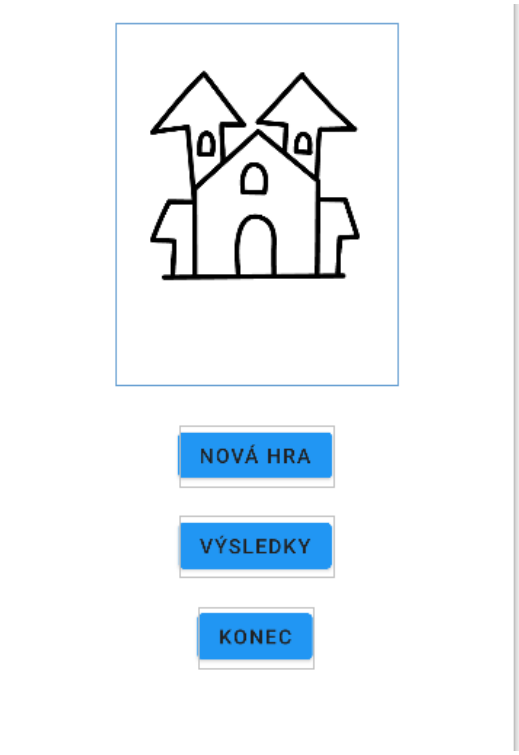
Styl si pojmenujeme a vybereme atributy, které chceme, aby ve stylu byly. Styl se nám uloží do složky resources/values do souboru styles.xml.



U tlačítka Výsledky upravíme atribut style na hodnotu @style/Tlacitko.

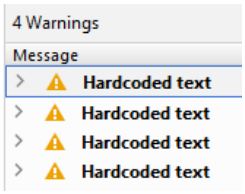
Stejným způsobem přidáme tlačítko s id **exitBtn** a nápisem **Konec**.

Všechy prvky správně ukotvíme. Obrazovka může vypadat například takto.



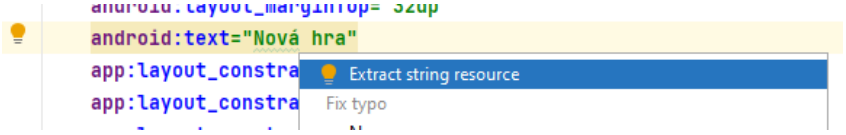
Zbývá nám poslední věc. Ve varování svítí varování Hardcoded text. Případně v XML je podsvícený řádek android:text. Je dobrou praxí, texty do activit nezadávat napřímo, ale odkazem na String.

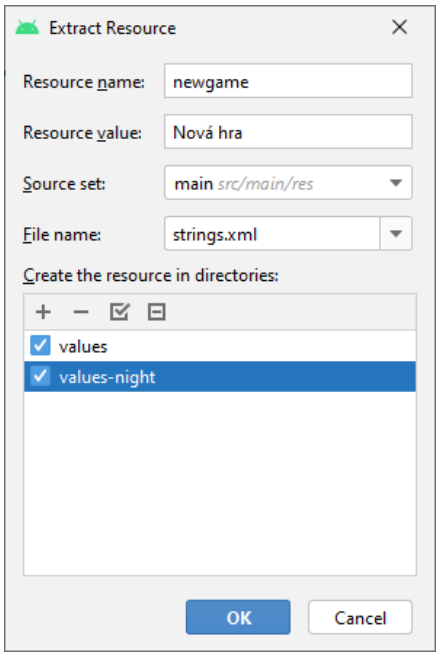
To v budoucnu umožní sjednotit pojmenování tlačítek se stejným chováním, jazykové mutace atd.




```
<Button
    android:id="@+id/newGameBtn"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="32dp"
    android:text="Nová hra"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/imageView"
    style="@style/Tlacitko" />
```

Řetězec opravíme v kontextovém menu (Alt+Enter) pomocí Extract string resouce. Kdy si řetězec pojmenujeme a uložíme do strings.xml.



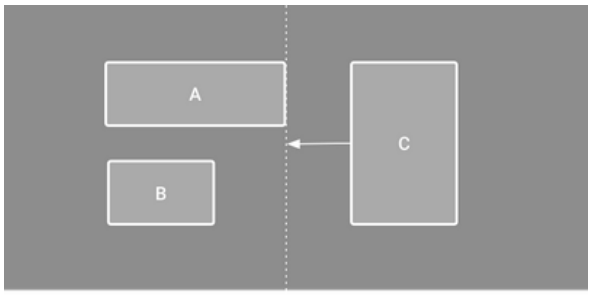


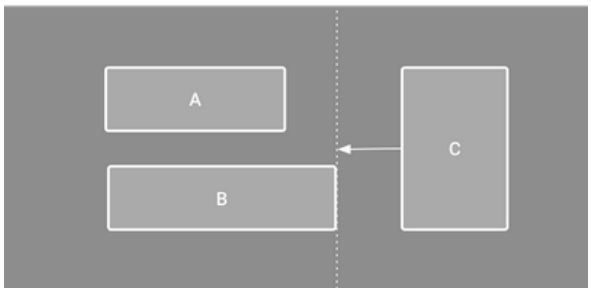
Před ukončením návrhu obrazovky je vhodné si ověřit, jak se bude chovat na různě velikých zařízeních, s různou rotací a zapnutým nočním režimem. Pak třeba zjistíte, že když mobil v noci otočíte a tak něco neuvidíte.

V component tree můžete označit více UI prvků a vytvořit z nich v kontextovém menu Horizontální nebo vertikální řetez.

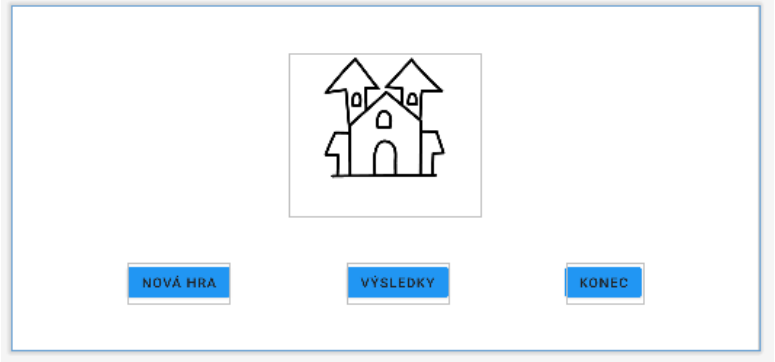
V tom samém místě může přidávat **Add helpers / Guideline** - na pevné pozici umístěnou čáru, ke které se přichytávají další prvky.

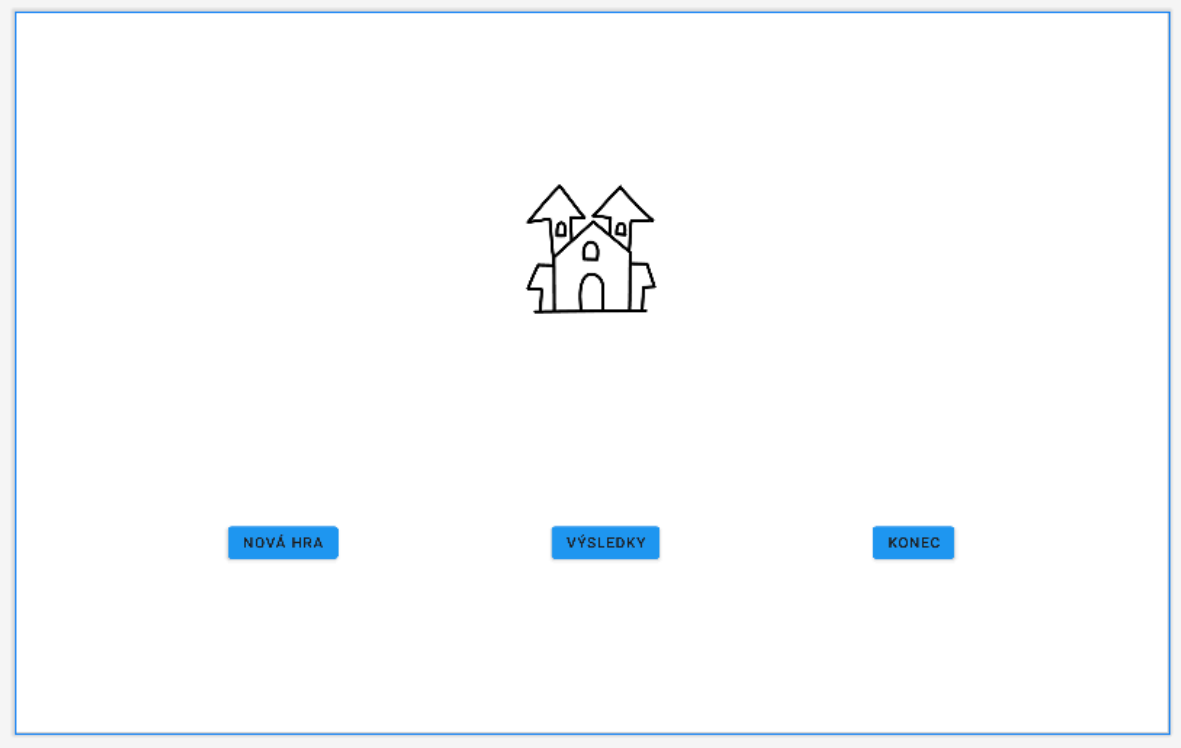
Nebo označením více UI prvků lze přidat **Add helpers/Barrier**. Pak se vytvoří čára, která se připne na maximální rozměr skupiny. Tím lze řešit problém, že velikost prvků se dynamicky mění (různé překlady popisků políček).





Vhodnou kombinací omezení lze dosáhnout, že obrazovka bude funkční na různých velikostí zařízení při různém otočení.

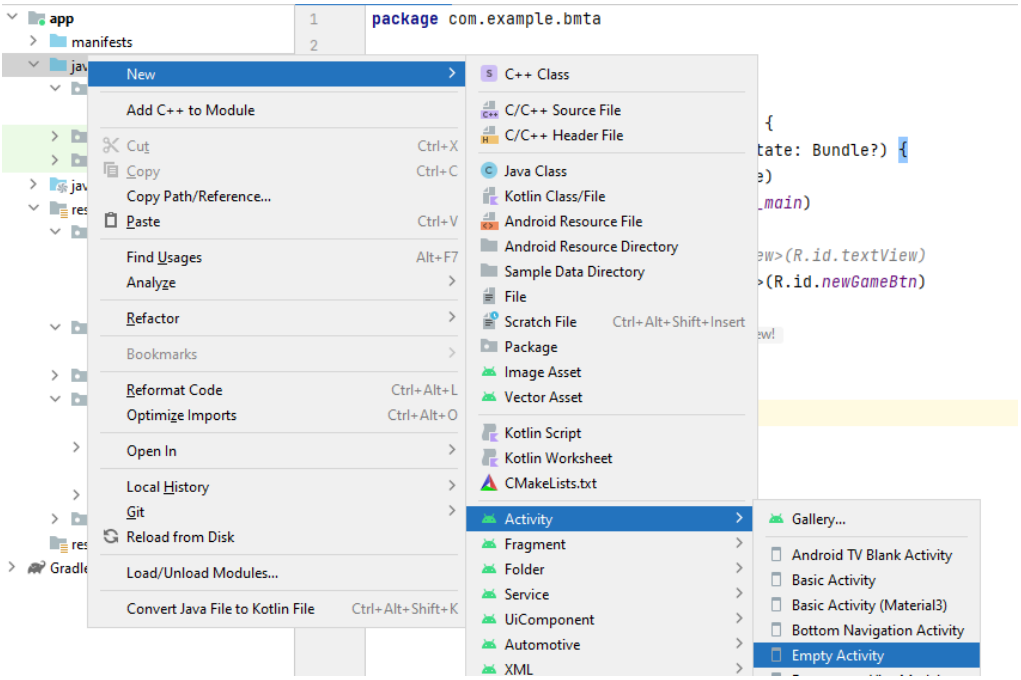




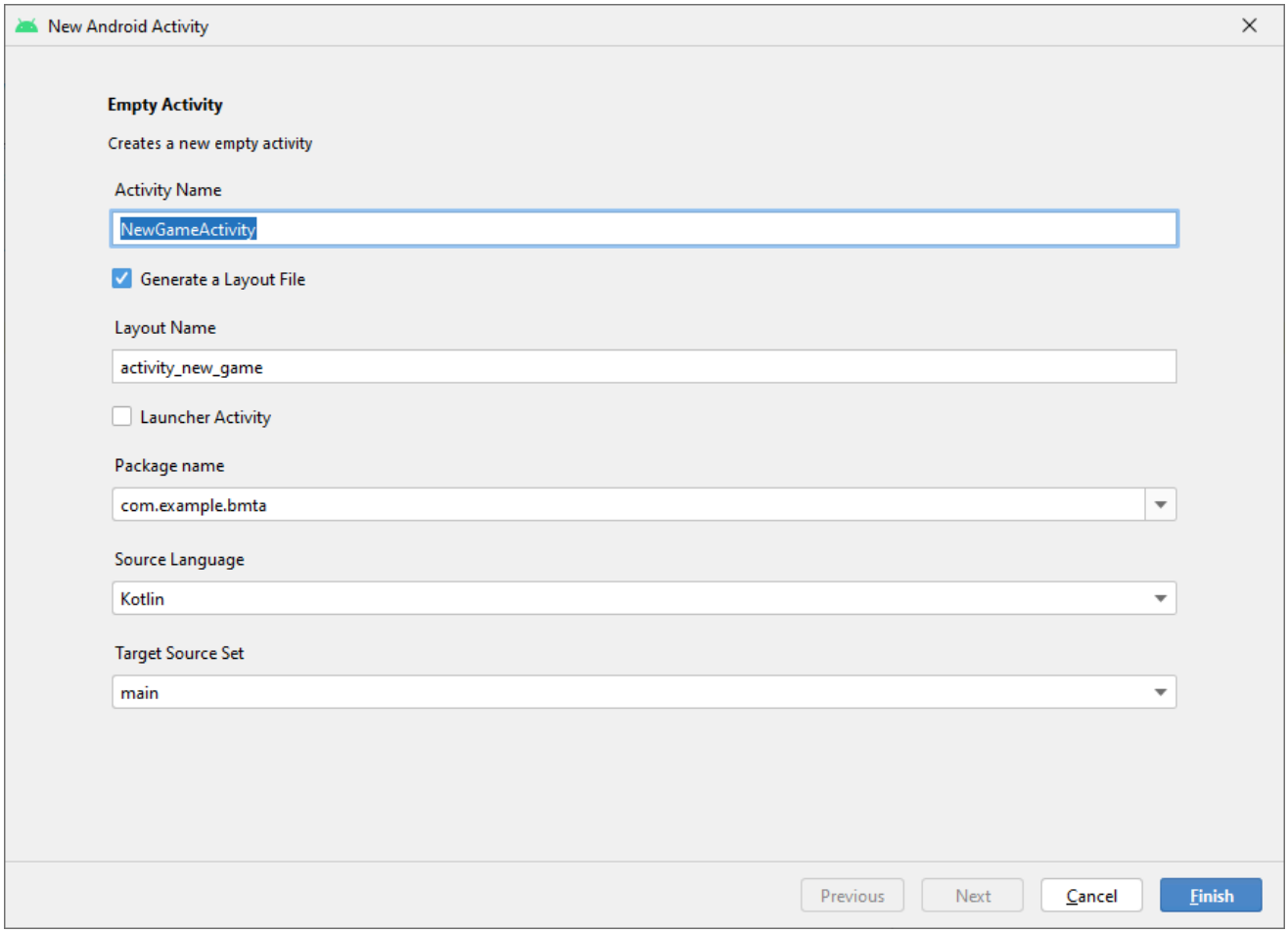
Někdy se musíme spokojit s tím, že některá zařízení podporovat nebudeme (hodinky).

Další aktivita

Vytvoříme si novou aktivitu, která se bude spouštět po kliknutí na tlačítko Nová hra. V této aktivitě hráč zadá své jméno.



Aktivitu pojmenujem **NewGameActivity**. Opět se nám vytvoří kotlin třída a XML soubor.



Zároveň je vhodné se podívat do souboru **manifest/AndroidManifest.xml**. Jedná se o soubor, který obsahuje popis naší aplikace. Aby aplikace mohla používat aktivitu, musí být zde uvedená.

```
<activity
    android:name=".NewGameActivity"
    android:exported="false">
    <meta-data
        android:name="android.app.lib_name"
        android:value="" />
</activity>
```

Do aktivity přidáme následující prvky

- **TextView, id=nameHeroTxt, text=Jméno hráče**
- **PlainText, id=heroNameEdit, hint=Hrdina**
- **Button, id=startBtn, text=Začít hrát, styl=...**

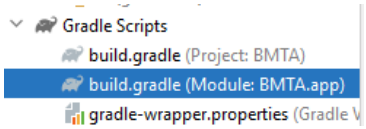
Vše řádně ukotvěte, vyřešte varování ohledně Hardcode text, případně barevné kontrasty, autofill apod. Obrázovka může vypadat například takto.

Přechod mezi obrazovkami a view binding

Odkazování pomocí findViewById je starý nepraktický způsob. Existuje více možností, jak interakci obrazovky a programu řešit.

- Data binding - Gradle na pozadí generuje kód (třídy), které když se změní hodnota proměnné v modelu, automaticky aktualizuje odpovídající UI prvek. Vhodné pro větší projekty, pro menší projekty možná až zbytečně komplikované.
- View binding - Gradle na pozadí generuje kód (třídy), které propojují XML soubor a odpovídající kotlin třídu.
- Jetpack compose - zcela odlišný návrh UI, který nepoužívá XML a designer, ale obrazovky se definují voláním metod knihovny přímo v kotlinu.

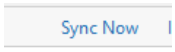
V tomto projektu budeme používat View binding. Který musíme nastavit. Otevřeme soubor build.gradle (Module BMTA.app)



Dopíšeme do sekce android následující kód.

```
buildFeatures {
    viewBinding = true
}
```

Nesmíme zapomenout spustit Sync Now, který znovu provede rebuild projektu a na pozadí vygeneruje pomocné třídy.



Otevřeme soubor MainActivity.kt a upravíme MainActivity, aby používala ViewBinding.

Proměnná binding používá na pozadí vygenerovanou třídu, jejíž jméno se odvíjí od jména aktivity.

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding : ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

Pokud chceme přistupovat k UI prvku z programu, nemusí si definovat nové proměnné, pouze použijeme binding.id_ui_prvku.

Například takto vypadá ošetření stisknutí tlačítka Nová hra.

binding.newGameBtb umožňuje přistoupit k atributům tlačítka.

Pro start nové aktivity potřebujeme vytvořit nový object Intent. Ten obecně slouží k popisu operace, která se má provést. this odkazuje na aktuální activitu, druhý parametr je jméno třídy nové aktivity.

```
binding.newGameBtn.setOnClickListener {
    startActivity(Intent(this, NewGameActivity::class.java))
}
```

Kód pro ukončení aplikace (minimalizace) vypadá následovně. Opět vytváříme Intent, ve kterém simulujeme stištění Andoid tlačítka Home.

```
binding.exitBtn.setOnClickListener {
    val intent = Intent(Intent.ACTION_MAIN)
    intent.addCategory(Intent.CATEGORY_HOME)
    startActivity(intent)
}
```

Upravíme si třídu NewGameActivity, aby podporovala View Binding.

```
class NewGameActivity : AppCompatActivity() {
    private lateinit var binding : ActivityNewGameBinding
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityNewGameBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

Git

Změny projektu commitneme do gitu a pusheme do vzdálené repository.

Naposledy změněno: čtvrtek, 2. května 2024, 15.35

◄ Vytvoření UI (úvod do Android Studia) - video

Přejít na...

Cvičení - vektorové obrázky (zip soubor) ►

Nápověda a dokumentace (anglicky)
Kontaktujte podporu stránek

Jste přihlášeni jako Lukáš Čegan (Odhlásit se)
DANTE_MA_FINAL

Moje stránka
Kalendář
Kontakty
Mahara
Kurzy
Čeština (cs)
Čeština (cs)
English (en)

Souhrn uchovávaných dat
Stáhněte si mobilní aplikaci

Životní cyklus aplikace

Komponenty, které si mají na starost životní cyklus, provádějí akce v reakci na změnu stavu životního cyklu jiné komponenty, například aktivit a fragmentů. Tyto komponenty pomáhají vytvářet lépe organizovaný a často lehčí kód, který se snadněji udržuje.

Běžným vzorem je implementace akcí závislých komponent do metod životního cyklu aktivit a fragmentů. Tento vzor však vede ke špatné organizaci kódu a k šíření chyb. Použitím komponent, které si uvědomují životní cyklus, můžete přesunout kód závislých komponent z metod životního cyklu do samotných komponent.

Na stránkách androidx.lifecycle je balíček, který poskytuje třídy a rozhraní, které umožňují vytvářet komponenty, jež *si uvědomují životní cyklus* - *což* jsou komponenty, které mohou automaticky upravovat své chování na základě aktuálního stavu životního cyklu aktivity nebo fragmentu.

Většina komponent aplikací definovaných ve frameworku Android je spojena s životními cykly. Životní cykly jsou spravovány operačním systémem nebo kódem frameworku, který běží ve vašem procesu. Jsou základem fungování systému Android a vaše aplikace je musí respektovat. Jejich nedodržení může způsobit únik paměti nebo dokonce pád aplikace.

Představte si, že máme aktivitu, která na obrazovce zobrazuje polohu zařízení. Běžná implementace může vypadat takto:

```
internal class MyLocationListener(
    private val context: Context,
    private val callback: (Location) -> Unit
) {

  fun start() {
    // connect to system location service
  }

  fun stop() {
    // disconnect from system location service
  }
}

class MyActivity : AppCompatActivity() {
  private lateinit var myLocationListener: MyLocationListener

  override fun onCreate(...) {
    myLocationListener = MyLocationListener(this) { location ->
      // update UI
    }
  }

  public override fun onStart() {
    super.onStart()
    myLocationListener.start()
    // manage other components that need to respond
    // to the activity lifecycle
  }
```

```
  public override fun onStop() {
    super.onStop()
    myLocationListener.stop()
    // manage other components that need to respond
    // to the activity lifecycle
  }
}
```

I když tato ukázka vypadá dobře, ve skutečné aplikaci nakonec budete mít příliš mnoho volání, která spravují uživatelské rozhraní a další komponenty v závislosti na aktuálním stavu životního cyklu. Správa více komponent představuje značné množství kódu v metodách životního cyklu, jako je např. onStart() a onStop()., což ztěžuje jejich údržbu.

Navíc není zaručeno, že se komponenta spustí dříve, než je činnost nebo fragment zastaven. To platí zejména v případě, že potřebujeme provést dlouhotrvající operaci, jako je například nějaká kontrola konfigurace v systému onStart(). To může způsobit závodní podmínku, kdy onStop(() skončí dříve než metoda onStart()a komponenta tak zůstane naživu déle, než je potřeba.

```
class MyActivity : AppCompatActivity() {
  private lateinit var myLocationListener: MyLocationListener

  override fun onCreate(...) {
    myLocationListener = MyLocationListener(this) { location ->
      // update UI
    }
  }

  public override fun onStart() {
    super.onStart()
    Util.checkUserStatus { result ->
      // what if this callback is invoked AFTER activity is stopped?
      if (result) {
        myLocationListener.start()
      }
    }
  }

  public override fun onStop() {
    super.onStop()
    myLocationListener.stop()
  }

}
```

Na stránkách androidx.lifecycle je balíček, který poskytuje třídy a rozhraní, které vám pomohou řešit tyto problémy odolným a izolovaným způsobem.

Životní cyklus

Životní cyklus je třída, která uchovává informace o stavu životního cyklu komponenty (například aktivity nebo fragmentu) a umožňuje ostatním objektům tento stav sledovat.

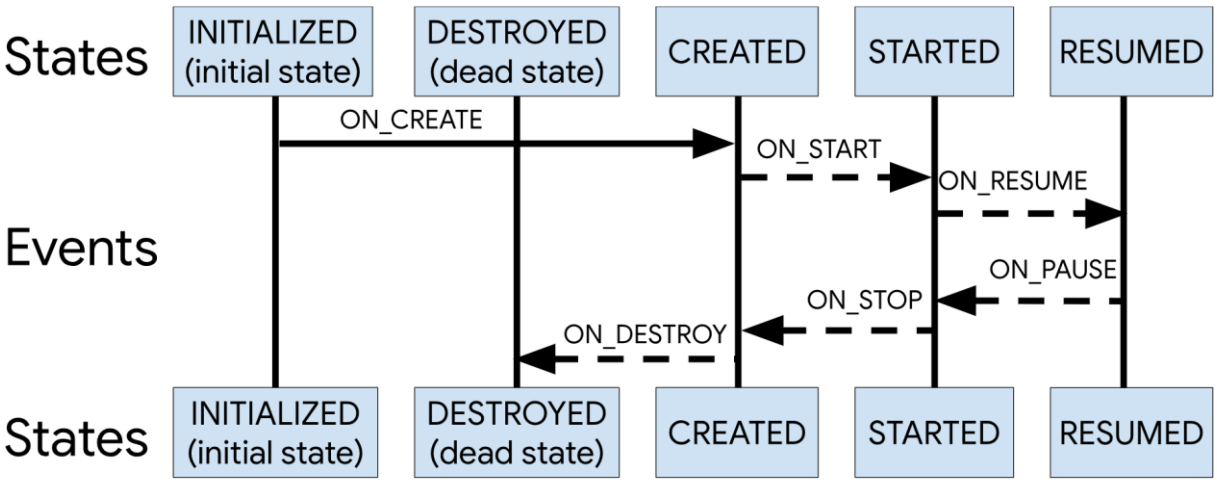
Životní cyklus používá dva hlavní výčty ke sledování stavu životního cyklu přidružené komponenty:

Událost

Události životního cyklu, které jsou odesílány z frameworku, a Životní cyklus třídy. Tyto události se mapují na události zpětného volání v aktivitách a fragmentech.

Stát

Aktuální stav komponenty sledované pomocí funkce Životní cyklus objektu.



Obrázek 1 - Stavy a události, které tvoří životní cyklus aktivity systému Android - převzato z oficiální dokumentace Android developer

Představte si stavy jako uzly grafu a události jako hrany mezi těmito uzly.

Třída může sledovat stav životního cyklu komponenty implementací funkce DefaultLifecycleObserver . a přepsáním odpovídajících metod, jako je onCreate, onStart atd. Pak lze přidat pozorovatele voláním metody addObserver() metody Lifecycle a předáte instanci svého pozorovatele, jak je ukázáno v následujícím příkladu:

```
class MyObserver : DefaultLifecycleObserver {
  override fun onResume(owner: LifecycleOwner) {
    connect()
  }

  override fun onPause(owner: LifecycleOwner) {
    disconnect()
  }
}

myLifecycleOwner.getLifecycle().addObserver(MyObserver())
```

Ve výše uvedeném příkladu implementuje objekt myLifecycleOwner funkci LifecycleOwner . rozhraní, které je vysvětleno v následující části.

LifecycleOwner

LifecycleOwner je rozhraní s jedinou metodou, které označuje, že třída má Lifecycle. Má jednu metodu, getLifecycle(), kterou musí třída implementovat. Pokud se místo toho snažíte spravovat životní cyklus celého aplikačního procesu, viz. ProcessLifecycleOwner.

Toto rozhraní abstrahuje vlastnictví životního cyklu od jednotlivých tříd, jako jsou Fragment a AppCompatActivitya umožňuje psát komponenty, které s nimi pracují. Jakákoli vlastní aplikační třída může implementovat rozhraní LifecycleOwner rozhraní.

Komponenty, které implementují DefaultLifecycleObserver . bezproblémově spolupracují s komponentami, které implementují LifecycleOwner protože vlastník může poskytnout životní cyklus, který může pozorovatel zaregistrovat ke sledování.

Pro příklad sledování polohy můžeme třídu MyLocationListener vytvořit tak, aby implementovala DefaultLifecycleObserver . a pak ji inicializovat pomocí aktivity Lifecycle v onCreate() metodu. To umožňuje, aby třída MyLocationListener byla soběstačná, což znamená, že logika reakce na změny stavu životního cyklu je deklarována v MyLocationListener namísto v aktivitě. Díky tomu, že si jednotlivé komponenty ukládají vlastní logiku, je správa aktivit a logiky fragmentů jednodušší.

```
class MyActivity : AppCompatActivity() {
  private lateinit var myLocationListener: MyLocationListener

  override fun onCreate(...) {
    myLocationListener = MyLocationListener(this, lifecycle) { location ->
      // update UI
    }
    Util.checkUserStatus { result ->
      if (result) {
        myLocationListener.enable()
      }
    }
  }
}
```

Běžným případem použití je vyhnout se vyvolání určitých zpětných volání, pokud je Životní cyklus není právě v dobrém stavu. Například pokud by zpětné volání spustilo transakci fragmentu po uložení stavu aktivity, vyvolalo by to pád, takže bychom toto zpětné volání nikdy nechtěli vyvolat.

Aby byl tento případ použití snadný, je Životní cyklus umožňuje ostatním objektům dotazovat se na aktuální stav.

```
internal class MyLocationListener(
    private val context: Context,
```

```
        private val lifecycle: Lifecycle,
        private val callback: (Location) -> Unit
): DefaultLifecycleObserver {

    private var enabled = false

    override fun onStart(owner: LifecycleOwner) {
        if (enabled) {
            // connect
        }
    }

    fun enable() {
        enabled = true
        if (lifecycle.currentState.isAtLeast(Lifecycle.State.STARTED)) {
            // connect if not connected
        }
    }

    override fun onStop(owner: LifecycleOwner) {
        // disconnect if connected
    }
}
```

Díky této implementaci je naše třída LocationListener zcela závislá na životním cyklu. Pokud potřebujeme použít náš LocationListener z jiné aktivity nebo fragmentu, stačí jej inicializovat. Veškeré operace nastavení a ukončení jsou řízeny samotnou třídou.

Pokud knihovna poskytuje třídy, které musí pracovat s životním cyklem systému Android, doporučujeme používat komponenty, které tento cyklus podporují. Klienti vaší knihovny mohou tyto komponenty snadno integrovat bez nutnosti ruční správy životního cyklu na straně klienta.

Implementace vlastního vlastníka životního cyklu LifecycleOwner

Fragmenty a aktivity v podpůrné knihovně 26.1.0 a novější již implementují funkci LifecycleOwner rozhraní.

Pokud máte vlastní třídu, ze které chcete vytvořit LifecycleOwner, můžete použít LifecycleRegistry ale musíte do této třídy předávat události, jak je uvedeno v následujícím příkladu kódu:

```
class MyActivity : Activity(), LifecycleOwner {

    private lateinit var lifecycleRegistry: LifecycleRegistry

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        lifecycleRegistry = LifecycleRegistry(this)
        lifecycleRegistry.markState(Lifecycle.State.CREATED)
    }

    public override fun onStart() {
```

```
        super.onStart()
        lifecycleRegistry.markState(Lifecycle.State.STARTED)
    }

    override fun getLifecycle(): Lifecycle {
        return lifecycleRegistry
    }
}
```

Osvědčené postupy pro komponenty s ohledem na životní cyklus

- Udržujte ovladače uživatelského rozhraní (aktivity a fragmenty) co nejjednodušší. Neměly by se snažit získávat vlastní data; místo toho použijte nástroj ViewModel a dodržujte LiveData a promítat změny zpět do pohledů.
- Snažte se psát uživatelská rozhraní řízená daty, kde je odpovědností řadiče uživatelského rozhraní aktualizovat zobrazení při změně dat nebo oznamovat akce uživatele zpět do řídicího systému. ViewModel.
- Umístěte datovou logiku do ViewModel třídy. ViewModel by měl sloužit jako spojovací článek mezi kontrolérem uživatelského rozhraní a zbytkem aplikace. Buďte však opatrní, není to ViewModeljeho odpovědností získávat data (například ze sítě). Namísto toho, ViewModel by měl zavolat příslušnou komponentu, aby data načetla, a poté poskytnout výsledek zpět kontroléru uživatelského rozhraní.
- Pomocí funkce Data Binding můžete zachovat čisté rozhraní mezi pohledy a řadičem uživatelského rozhraní. To vám umožní vytvořit pohledy více deklarativní a minimalizovat aktualizační kód, který musíte psát v aktivitách a fragmentech. Pokud to raději děláte v programovacím jazyce Java, použijte knihovnu, jako je např. Butter Knife abyste se vyhnuli šablonovitému kódu a měli lepší abstrakci.
- Pokud je vaše uživatelské rozhraní složité, zvažte vytvoření prezentér který bude zpracovávat úpravy uživatelského rozhraní. Může to být pracné, ale může to usnadnit testování komponent uživatelského rozhraní.
- Vyhněte se odkazování na Zobrazit nebo Aktivita v kontextu ViewModel. Pokud ViewModel přežije aktivitu (v případě změn konfigurace), vaše aktivita unikne a není správně zlikvidována garbage collectorem.
- Ke správě dlouhotrvajících úloh a dalších operací, které mohou probíhat asynchronně, použijte koroutiny jazyka Kotlin.

Případy použití pro komponenty s ohledem na životní cyklus

Komponenty, které si uvědomují životní cyklus, vám mohou v různých případech výrazně usnadnit správu životních cyklů. Několik příkladů:

- Přepínání mezi hrubými a jemnými aktualizacemi polohy. Pomocí komponent, které si uvědomují životní cyklus, můžete povolit jemné aktualizace polohy, když je aplikace pro určování polohy viditelná, a přepnout na hrubé aktualizace, když je aplikace na pozadí. LiveData, komponenta, která si uvědomuje životní cyklus, umožňuje aplikaci automaticky aktualizovat uživatelské rozhraní, když uživatel změní polohu.

- Zastavení a spuštění vyrovnávací paměti videa. Pomocí komponent, které si uvědomují životní cyklus, spusťte vyrovnávací paměť videa co nejdříve, ale přehrávání odložte až do úplného spuštění aplikace. Komponenty sledující životní cyklus můžete také použít k ukončení vyrovnávací paměti, když je aplikace zničena.
- Spuštění a zastavení síťového připojení. Pomocí komponent, které si uvědomují životní cyklus, lze umožnit živou aktualizaci (streamování) síťových dat, když je aplikace v popředí, a také automatické pozastavení, když aplikace přejde do pozadí.
- Pozastavení a obnovení animovaných kreslených objektů. Pomocí komponent, které si uvědomují životní cyklus, lze animované kreslitelné objekty pozastavit, když je aplikace na pozadí, a obnovit kreslitelné objekty poté, co se aplikace dostane do popředí.

Zpracování událostí zastavení

Když se Životní cyklus patří do AppCompatActivity nebo fragmentu, je Lifecyclese změní na CREATED a událost ON_STOP se odešle, když se Cyklu AppCompatActivity nebo Fragment's onSaveInstanceState(). je zavolána.

Když fragment nebo AppCompatActivityje stav uložen prostřednictvím onSaveInstanceState()., je jeho uživatelské rozhraní považováno za neměnné, dokud není zavolána funkce ON_START. Pokus o změnu uživatelského rozhraní po uložení stavu pravděpodobně způsobí nekonzistenci navigačního stavu aplikace, což je důvod, proč se aplikace FragmentManager vyhodí výjimku, pokud aplikace spustí FragmentTransaction po uložení stavu. Viz commit() pro podrobnosti.

Služba LiveData tomuto okrajovému případu zabraňuje tím, že se zdrží volání svého pozorovatele, pokud je přidružený pozorovatel. Životní cyklus není alespoň SPUŠTĚN. V zákulisí volá isAtLeast() předtím, než se rozhodne zavolat svého pozorovatele.

Bohužel, AppCompatActivity's onStop() je volána až po onSaveInstanceState()., což ponechává mezeru, ve které nejsou povoleny změny stavu uživatelského rozhraní, ale Životní cyklus ještě nebyl přesunut do stavu CREATED.

Aby se tomuto problému předešlo, je Životní cyklus ve verzi beta2 a nižší označit stav jako CREATED bez odeslání události, takže jakýkoli kód, který kontroluje aktuální stav, získá skutečnou hodnotu, i když událost není odeslána, dokud není odeslána onStop() je zavolána systémem.

Toto řešení má bohužel dva zásadní problémy:

- Na úrovni API 23 a nižší systém Android skutečně ukládá stav aktivity, i když je *částečně* pokryta jinou aktivitou. Jinými slovy, systém Android volá onSaveInstanceState() ale nemusí nutně volat onStop(). Tím vzniká potenciálně dlouhý interval, kdy si pozorovatel stále myslí, že životní cyklus je aktivní, přestože stav jeho uživatelského rozhraní nelze změnit.

- Každá třída, která chce vystavit podobné chování jako třída LiveData musí implementovat řešení, které poskytuje třída Lifecycle verze beta 2 a nižší.

Mobilní aplikace

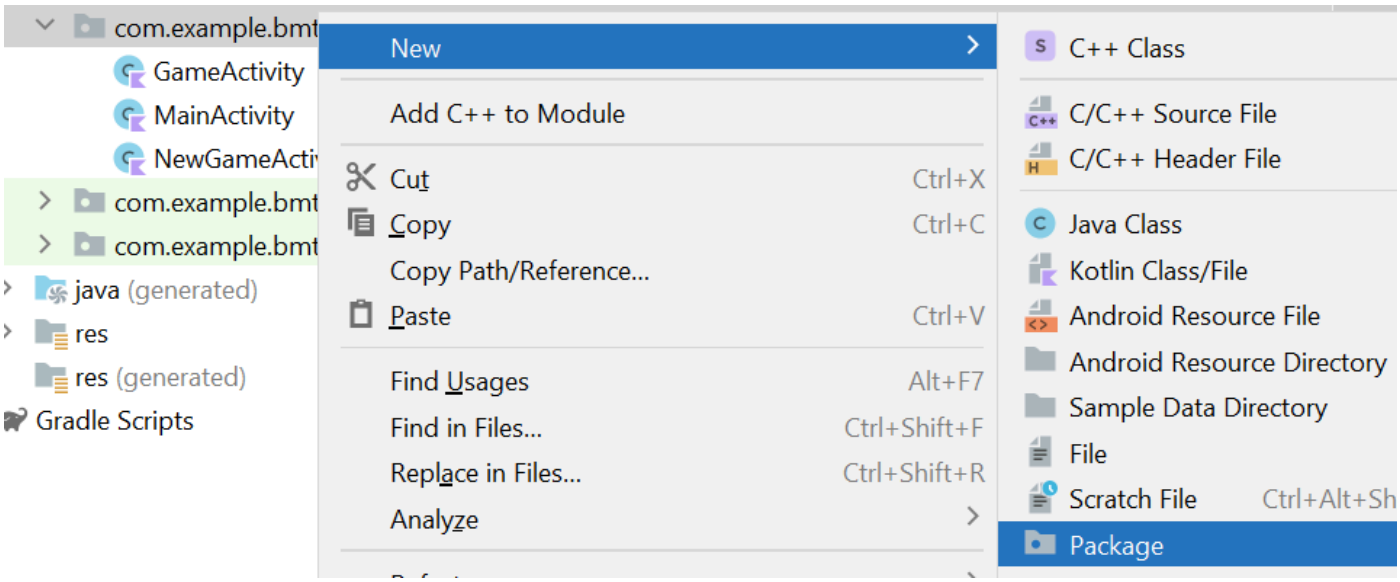
Propojení UI a modelu (MVC, view binding, recycler view) - text

Cílem tohoto cvičení je nahradit textové uživatelské prostředí hry za UI vytvořené předchozí cvičení.

Architektura MVC

Pro začátek použijeme architekturu Model View Controller.

V projektu si vytvoříme packages **view**, **model** a **contoller**.



Aktivity Game, Main a NewGame přesuneme do package **view**.

Do package **model** nakopírujeme soubory

- Character.kt
- Direction.kt
- Enemy.kt
- GameField.kt
- GameObject.kt
- GamePlan.kt
- Hero.kt
- Item.kt
- Position.kt
- Terrain.kt

Do package controller nakopírujeme soubor

- Game.kt

Úpravy modelu

Do třídy **Direction** si přidáme ještě jednu hodnotu pro žádný směr (NOMOVE).

```
enum class Direction (val relativeY : Int, val relativeX : Int, val description: String, val command: String) {
    NORTH (-1, 0, "sever", "s"),
    SOUTH (1, 0, "jih", "j"),
    EAST (0, 1, "východ", "v"),
    WEST (0, -1, "západ", "z"),
    NORTHEAST (-1, 1, "severovýchod", "sv"),
    SOUTHEAST (1, 1, "jihovýchod", "jv"),
    NORTHWEST (-1, -1, "severozápad", "sz"),
    SOUTHWEST (1, -1, "jihozápad", "jz"),
    NOMOVE (0, 0, "nop", "nop")
}
```

Propojení aktivity Game s controllerem

V aktivitě **GameActivity** budeme potřebovat dvě proměnné. Jednu pro jméno Hrdiny a druhou pro samotnou hru.

```
private lateinit var game : Game
private var heroName : String = "hrdina"
```

V rámci metody **onCreate** získáme jméno hrdiny, které jsme načetli v aktivitě NewGame a zapsali jsme ho to Intent.

```
if (intent.extras != null) {
    heroName = intent.extras!!.getString("heroName").toString()
}
```

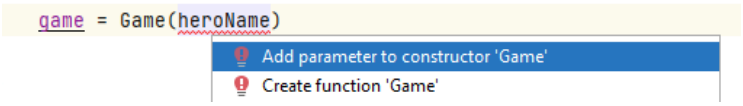
Ve chvíli, kdy máme jméno hrdiny, můžeme vytvořit instanci hry.

```
game = Game(heroName)
```

Textová hra neměla zcela správně navrženou MVC. Model jsem měli oddělený, ale controller a view byly sloučeny v rámci třídy Game. Proto budeme muset dělat postupné úpravy třídy Game, aby se z ní stal plnohodnotný controller a neobsahovala části, které komunikují s hráčem.

V rámci třídy se volala metoda readHeroName, která se používala pro vytvoření hrdiny. Jméno hrdiny máme z aktivity NewGame a předáme ho v rámci konstruktoru.

Stačí označit chybný řádek a zvolit Add parameter to construktor 'Game'.



Ve třídě Game pak upravíme vytváření hrdiny a metodu readHeroName můžeme zakomentovat nebo smazat.

```
var hero = Hero (name= heroName, position = gamePlan.generateRandomPositionOnMeadow())
```

Na začátku hry je vhodné aktualizovat UI prvky, které zobrazují statistiku hráče. V aktivitě GameActivity si vytvoříme funkci **refreshHeroStats**, kterou zavoláme z metody onCreate.

```
private fun refreshHeroStats () {
    binding.textHeroName.text = game.hero.name
    binding.textScore.text = game.score.toString()
    binding.textHealth.text= "%.2f".format(game.hero.health)
    binding.textAttack.text= "%.2f".format(game.hero.attack)
    binding.textDefense.text= "%.2f".format(game.hero.defense)
    binding.textHealing.text= "%.2f".format(game.hero.healing)
    binding.textKills.text = game.hero.kills.toString()
}
```

Aplikaci můžeme spustit a zkontrolovat, že opravdu se UI prvky změnily.

Aktualizace herních polí

Ve třídě Game máme metodu **getSurroundingDescription**, která v textové podobě vypisuje okolí, které hrdina vidí. Opět tato metoda patří do view a ne do controlleru. Navíc metoda nevracela jednoduchou hodnotu, ale komplikovaný string s celým okolím.

Budeme ji muset patřičně nahradit. Vytvoříme metodu **getObjectOnPosition** ve třídě Game. Která bude mít jako vstupní parametr směr, kterým se hrdina dívá. Výstupem bude řetězec, který popíše, co daným směrem hrdina vidí.

- Pokud je herní pole prázdné, vrátí jméno terénu
- Pokud je na herním poli předmět, vrátí jeho jméno
- Pokud je na herním poli živý nepřítel, vrátí jeho jméno
- Pokud je na herním poli mrtvola, vrátí slovo hrob

Nejprve si vytvoříme novou pozici na základě pozice hrdiny a směru. Zde se nám bude hodit hodnota NOMOVE, abychom se podívali i na pozici, na které je hrdina.

Ve třídě Game máme již metody getEnemyOnGameField a getItemOnGameField, tak je použijeme.

```
fun getObjectOnPosition (direction: Direction): String {
    val newPosition = Position(hero.position, direction)
    val enemy = getEnemyOnGameField(newPosition, enemies)
    val item = getItemOnGameField(newPosition, items)

    if (enemy is Enemy) {
        return if (enemy.isDead()) {
            "Hrob"
        } else {
            enemy.name
        }
    } else if (item is Item && ! item.pickedUp) {
        return item.name
    }

    return gamePlan.getGameField(newPosition).terrain.toString()
}
```

Metoda **getObjectOnPosition** nám vrátí řetezec popisující objekt daným směrem. Místo řetězce budeme chtít hráči zobrazit ikonu. Musíme mít něco, co převede řetězec na obrázek.

Vytvoříme si **enums Icons**, který řetězci přiřazuje odkaz do res/drawable/jmeno souboru

```
import com.example.bmta.R

enum class Icons (val imgResource: Int){
    BORDER (R.drawable.border),
    MEADOW (R.drawable.meadow),
    FOREST (R.drawable.forest),
    RIVER (R.drawable.river),
    BRIDGE (R.drawable.bridge),
    Skeleton (R.drawable.skeleton),
    Hrob (R.drawable.rip),
    Troll (R.drawable.troll),
    Mec (R.drawable.sword),
    Stit (R.drawable.shield),
    Dyka (R.drawable.knife),
    Helma (R.drawable.helmet),
    Brneni (R.drawable.armor),
    Lekarna (R.drawable.medical)
}
```

Nyní v v aktivitě GameActivity vytvoříme metodu **refreshGameFields**. Metoda pro každý obrázek zjistí, co vidí hrdina daným směrem (**getObjectOnPosition**). Řetězec se dohledá v Icons a zjistí se obrázek.

Metodu zavoláme v metodě onCreate.

```
private fun refreshGameFields() {
    binding.imageNorth.setImageResource(
        Icons.valueOf(game.getObjectOnPosition(Direction.NORTH)).imgResource)

    binding.imageNorthEast.setImageResource(
        Icons.valueOf(game.getObjectOnPosition(Direction.NORTHEAST)).imgResource)

    binding.imageEast.setImageResource(
        Icons.valueOf(game.getObjectOnPosition(Direction.EAST)).imgResource)

    binding.imageSouthEast.setImageResource(
        Icons.valueOf(game.getObjectOnPosition(Direction.SOUTHEAST)).imgResource)

    binding.imageSouth.setImageResource(
        Icons.valueOf(game.getObjectOnPosition(Direction.SOUTH)).imgResource)

    binding.imageSouthWest.setImageResource(
        Icons.valueOf(game.getObjectOnPosition(Direction.SOUTHWEST)).imgResource)

    binding.imageWest.setImageResource(
        Icons.valueOf(game.getObjectOnPosition(Direction.WEST)).imgResource)

    binding.imageNorthWest.setImageResource(
        Icons.valueOf(game.getObjectOnPosition(Direction.NORTHWEST)).imgResource)

    binding.imageCenter.setImageResource(
        Icons.valueOf(game.getObjectOnPosition(Direction.NOMOVE)).imgResource)
}
```

Ošetření uživatelských vstupů

Hru bude hráč ovládat klikáním na obrázky. Pro každý obrázek vytvoříme Listner, který zavolá metodu move a předají jako jediný parametr směr. Listenery umístíme do funkce **onCreate**.

```
binding.imageNorth.setOnClickListener {runRound(Direction.NORTH)}
binding.imageNorthEast.setOnClickListener {runRound(Direction.NORTHEAST)}
binding.imageEast.setOnClickListener {runRound(Direction.EAST)}
binding.imageSouthEast.setOnClickListener {runRound(Direction.SOUTHEAST)}
binding.imageSouth.setOnClickListener {runRound(Direction.SOUTH)}
binding.imageSouthWest.setOnClickListener {runRound(Direction.SOUTHWEST)}
binding.imageWest.setOnClickListener {runRound(Direction.WEST)}
binding.imageNorthWest.setOnClickListener {runRound(Direction.NORTHWEST)}
binding.imageCenter.setOnClickListener {runRound(Direction.NOMOVE)}
```

V listnerech voláme metodu **runRound**, kterou ještě nemáme. Nezbývá než tuto metodu napsat. Bude se vlastně jednat o ekvivalent původní metody run ve třídě Game. Původní metoda run opět porušovala architekturu MVC, protože kombinovala čtení a ošetřování uživatelských vstupů (readCommand, checkCommand, setPossibleCommands) s vlastní logikou hru. Zároveň s přechodem z textové aplikace do aplikace řízené událostmi se zcela mění logika algoritmu. V programu není hlavní cyklus, ale ošetřování uživatelských vstupů, které vyvolávají akce.

Vytvoříme metodu **runRound** se vstupním parametrem **direction**. Metoda provede kolo hry (**game**.**runRound**). Případnou zprávu zobrazí na obrazovce pomocí **Toast**.

Pak otestuje, zda hra neskončila. Pokud ano, tak vyvolá Alert dialog a po stisknutí tlačítka Ok nastartuje aktivitu MainActivity.

Před ukončením tahu je třeba aktualizovat UI prvky.

```
private fun runRound(direction: Direction) {
    var message = game.runRound(direction)
    if (message.isNotEmpty()) {
        Toast.makeText(this, message, Toast.LENGTH_SHORT).show()
    }

    message = game.isGameFinish()
    if (message.isNotEmpty()) {
        AlertDialog.Builder(this)
            .setCancelable(false)
            .setTitle("Konec hry")
            .setMessage(message)
            .setPositiveButton("OK") { _, _ ->
                startActivity(Intent(this, MainActivity::class.java))
            }
            .setIcon(android.R.drawable.ic_dialog_alert)
            .show()
    }

    refreshHeroStats()
    refreshGameFields()
}
```

Třída Game zatím žádnou metodu **runRound** neobsahuje. Tak ji vytvoříme.

Metoda podle směru a polohy hrdiny zjistí, jaký příkaz se má spustit getCommand. Příkaz provede. Případně nechá zaútočit nepřítele a nakonec uzdraví hrdinu.

Zprávy spojuje přes StringBuilder a vrací zpět.

```
fun runRound (direction : Direction) : String {
    val message = StringBuilder()
    val command = getCommand (direction)

    if (command != "") {
        message.append(runCommand(command))
    }
    message.append(enemyAttack())
    heroHealing()
    return message.toString()
}
```

Ve třídě Game nám ještě chybí metoda **getCommand**, která připomíná metodu setPossibleCommands. Metoda vrací příkaz, který se má provést podle sousedního herního políčka, kde je hrdina.

Na středním herním poli (směr NOMOVE), lze provést sběr předmětu nebo útok na živého nepřítele.

Na okolní pole s terénem louka nebo most lze provést pohyb. Na okolní pole s terén les lze provést příkaz kacej.

V ostatních případech metoda vrátí prázdný řetězec.

```
fun getCommand(direction: Direction) : String {
     if (direction == Direction.NOMOVE) {
        val enemy = getEnemyOnGameField(hero.position, enemies)
        val item = getItemOnGameField(hero.position, items)

        if (enemy is Enemy && !enemy.isDead()) {
            return "utok"
        } else if (item is Item && !item.pickedUp) {
            return item.name.lowercase()
        }
    } else {
        when (gamePlan.getGameField(Position(hero.position, direction)).terrain) {
            Terrain.MEADOW, Terrain.BRIDGE -> return direction.command
            Terrain.FOREST -> return "kacej"+direction.command
            Terrain.RIVER, Terrain.BORDER -> return ""
        }
    }
    return ""
}
```

Ve třídě Game po refactoringu nebudeme potřebovat metody **run**, **readCommand**, **checkCommand**, **setPossibleCommands**.

Z runCommand můžeme odebrat voláním map a metodu map v GamePlan také zrušit. Opět porušovala MVC.

Hru vyzkoušíme.

Lepší zobrazení zpráv

Po každém tahu se zobrazuje zpráva o provedeném příkazu. Sice jsme nastavili, že se má zobrazovat po krátkou dobu, ale i tak tento způsob zobrazení nestíhá rychlé tahy.

Zobrazované zprávy budeme ukládat do **LinkedList** pojmenovaného **logs**.

Ve třídě GameActivity vytvoříme tento seznam.

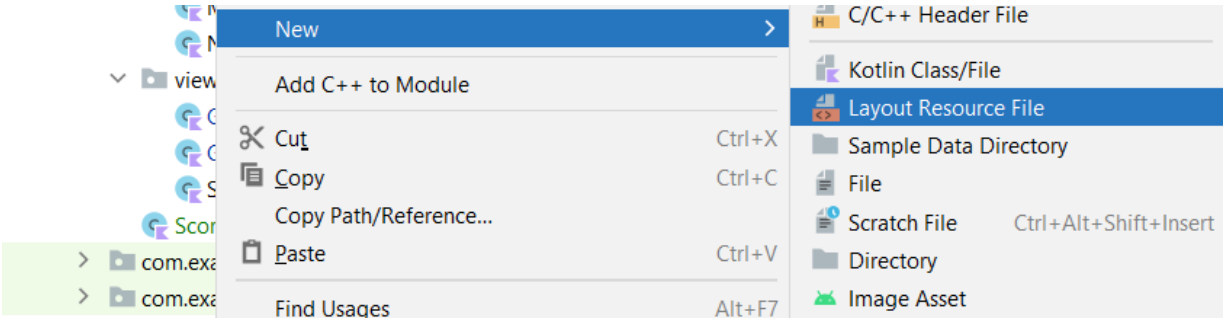
```
private val logs = LinkedList(listOf(""))
```

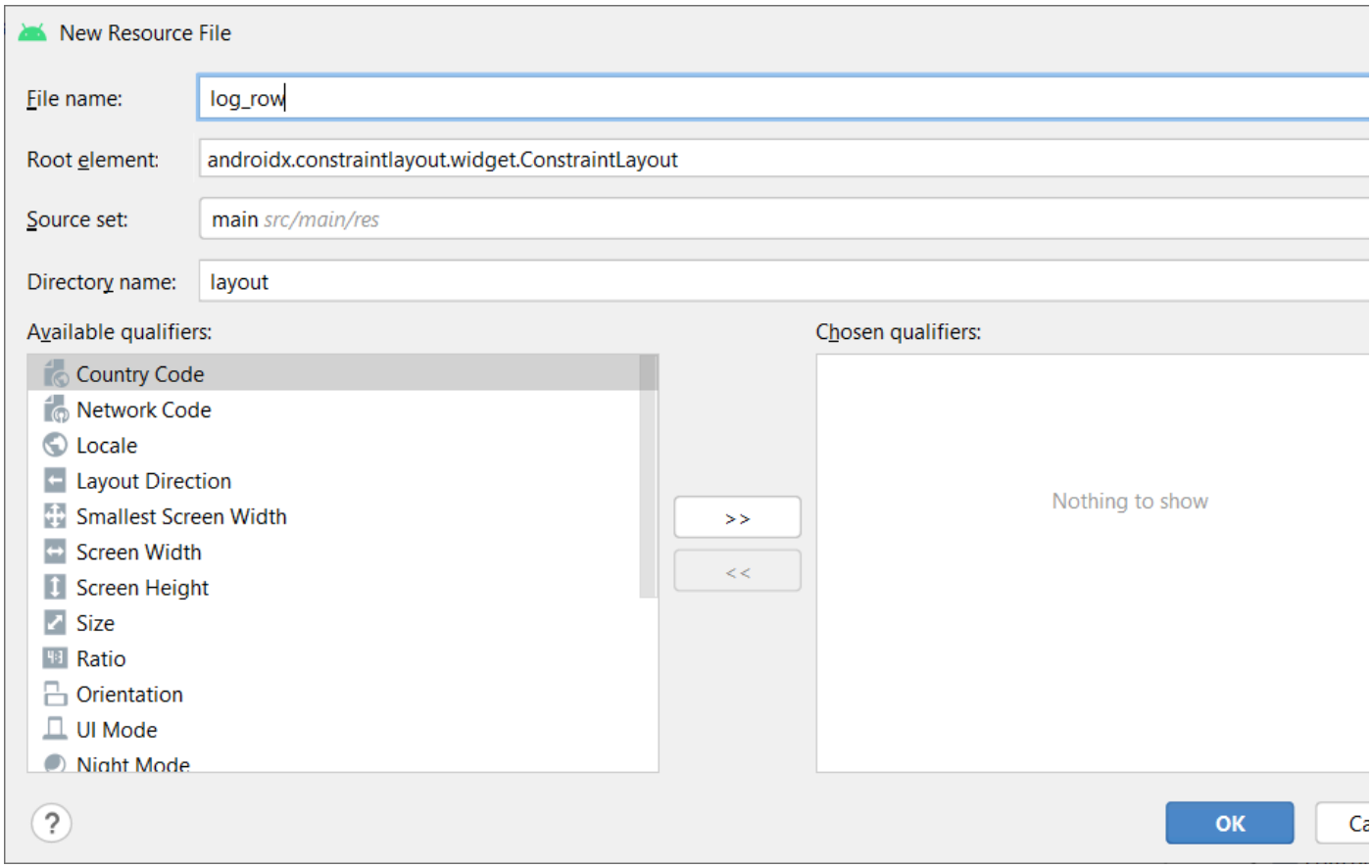
Volání Toast nahradíme za vložení zprávy do listu.

```
logs.push (message)
//Toast.makeText(this, message, Toast.LENGTH_SHORT).show()
```

Seznam budeme zobrazovat v **Recyler view**.

Nejprve si vytvoříme nový constraint_layout **log_row**, která bude představovat jeden řádek listu.



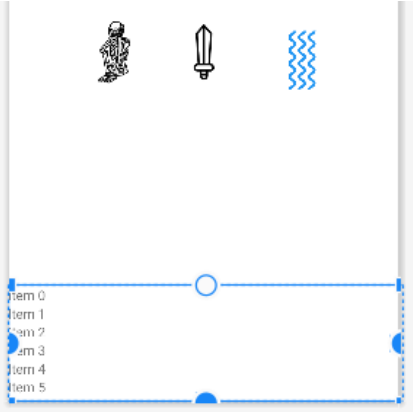


Do layoutu vložíme textView.

- id mu nastavíme na **row**.
- Textview ukotvíme nahoru, doleva, doprava.
- **layout_widht** nastavíme na 0dp - match constraint. Tím se prvek roztáhne na celou šířku.

Důležité je nastavit **u hlavního layoutu - layout_height na wrap content**. Pokud to neuděláme bude jeden záznam zobrazený na celé výšce recycler view.

Do aktivity GameActivity vložíme Recycler view a ukotvíme ho na spodní část obrazovky. id mu nastavíme na logRecycler.



V package view vytvoříme novou třídu **LogHolder**.

Proměnná textLog obsahuje odkaz na UI prvek, kde se bude zobrazovat obsah záznamu. textLog si pomocí findViewById inicializujeme v init bloku.

Připravíme si metodu setLogData, která aktualizuje řetězcem záznamu daný UI.

```
class LogHolder(view: View) : RecyclerView.ViewHolder(view) {
    private var textLog : TextView?

    init {
        textLog = view.findViewById(R.id.row)
    }

    fun setLogData(log: String) {
        textLog?.text = log
    }
}
```

V package view vytvoříme třídu **LogAdapter**.

Vstupem bude List logů. Protože třída dědí z RecyclerView.Adapter, musí mít metody

- getItemCount vrací celou velikost seznamu
- onBindViewHolder, která zajistí zobrazení i-tého záznamu seznamu pomocí holder
- onCreateViewHolder, která zajistí zobrazení activity_log_row v prostoru Recycler View.

```
class LogAdapter (val logList : List<String>) : RecyclerView.Adapter<LogHolder> () {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): LogHolder {
        return LogHolder(LayoutInflater.from(parent.context).inflate(R.layout.log_row, parent, false))
    }

    override fun onBindViewHolder(holder: LogHolder, position: Int) {
        holder.setLogData(logList[position])
    }

    override fun getItemCount(): Int {
        return logList.size
    }
}
```

V aktivitě GameActivity do metody onCreate musíme přidat inicializace recycler view.

Recycler view bude používat jednodušší lineární layout manager (porovnej s constraint layout).

Dále UI prvku musíme přiřadit připravený adapter.

```
binding.logRecycler.layoutManager = LinearLayoutManager(this)
binding.logRecycler.adapter = LogAdapter(logs)
```

Protože používáme view binding a ne data binding, neexistuje propojení mezi daty a UI prvky. Pokud se data změní, musí UI prvek informovat, že se má obvovit. To provedeme následujícím příkazem, který vložíme ze příkaz logs.push (message) do metody **runRound** ve třídě GameActivity.

```
binding.logRecycler.adapter?.notifyDataSetChanged()
```

Hru vyzkoušíme.

Git

Hru otestujeme a pokud vše funguje potvrdíme v gitu a pošleme do vzdáleného repository.

Naposledy změněno: čtvrtek, 2. května 2024, 15.33

◄ Propojení UI a modelu (MVC, view binding, recycler view) - video

Přejít na...

Cvičení - úkol (inventář předmětů) ►

Nápověda a dokumentace (anglicky)

Kontaktujte podporu stránek

Jste přihlášeni jako Lukáš Čegan (Odhlásit se)
DANTE_MA_FINAL

Moje stránka
Kalendář
Kontakty
Mahara
Kurzy
Čeština (cs)
Čeština (cs)
English (en)

Souhrn uchovávaných dat

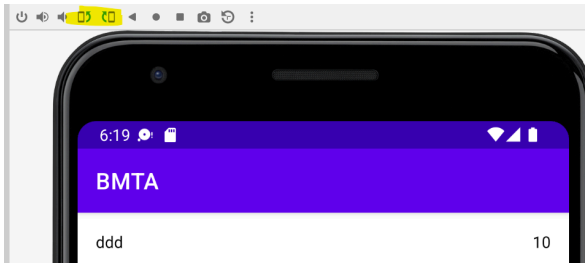
Stáhněte si mobilní aplikaci

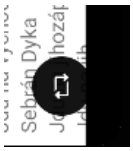
Mobilní aplikace

MVVM, nastavení v JSON - text

Model View ViewModel

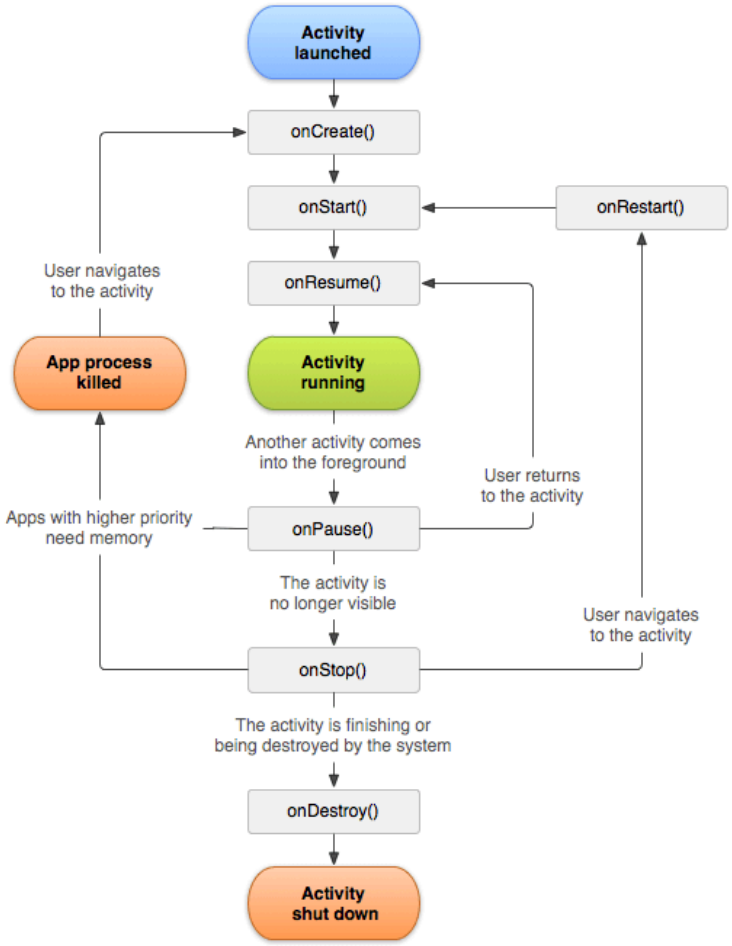
Spustíme hru a odehrajeme několik tahů. Pak provedeme rotaci telefonu a rotaci potvrdíme.





V tu chvíli dojde ke ztrátě rozehrané hry a jejímu restartu.

Důvodem je životní cyklus Activity. Ve chvíli, kdy dojde k otočení zařízení je activita znovu vytvořena a spuštěna a zavolána metoda onCreate.



V rámci metody onCreate je vytvoření nové instance třídy Game.

```
game = Game(heroName)
```

Tento problém je vyřešen zaměnením architektury MVC za **MVVM**. V androidu existuje třída **ViewModel**, která řeší přežití datového modelu spojeného s UI aktivitou během změn v jejím životním cyklu.

Třída Game bude dědit ze třídy **ViewModel**.

```
class Game(heroName: String) : ViewModel()
```

Zároveň budeme muset změnit vytváření instance modelu hry v aktivitě v metodě **GameActivity**. Pro vytváření bude potřeba vytvořit třídu **GameFactory**, která bude dědit z **ViewModelProvider.Factory**.

Konstruktor třídy bude obsahovat parametry, které se předají do konstruktoru třídy Game. Aktuálně se jedná o parametr heroName.

Dále je třeba přetížit metody create následujícím způsobem, aby vytvořila instanci třídy Game.

GameFactory nám zajistí, že pokud není vytvořena instance Game, tak ji vytvoří. Pokud již existuje, tak ji uchová a vrátí ji do aktivity během jejího vytváření.

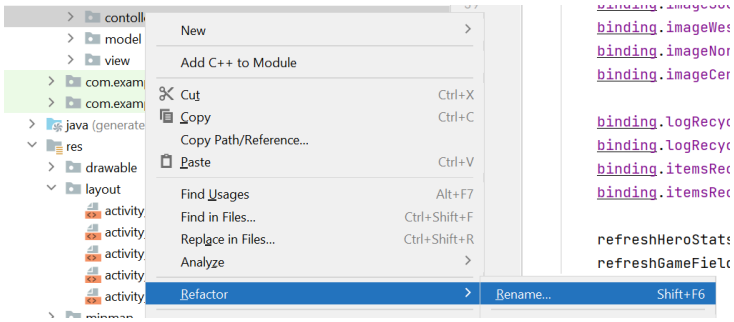
```
class GameFactory (
    private val heroName : String
): ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(Game::class.java)) {
            return Game(heroName) as T
        }
        throw IllegalArgumentException("Unknown Viewmodel class")
    }
}
```

Posledním krokem je vytvoření instance Game pomocí GameFactory v metodě onCreate aktivity GameActivity.

```
val factory = GameFactory(heroName)
game = ViewModelProvider(this, factory).get(Game::class.java)
```

Nyní můžeme aplikaci znovu spustit a vyzkoušet, že překlopení zařízení hra přežije bez restartu.

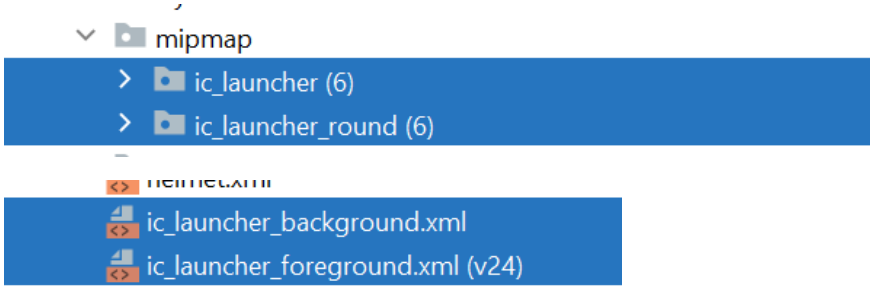
Package Controller přejmenujeme na viewmodel



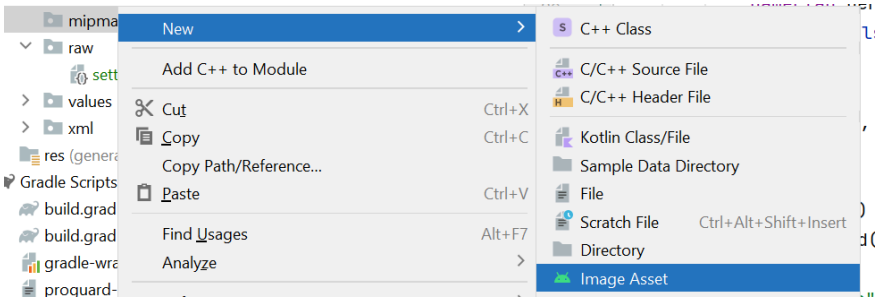
Ikona aplikace

Změníme výchozí ikonu aplikace.

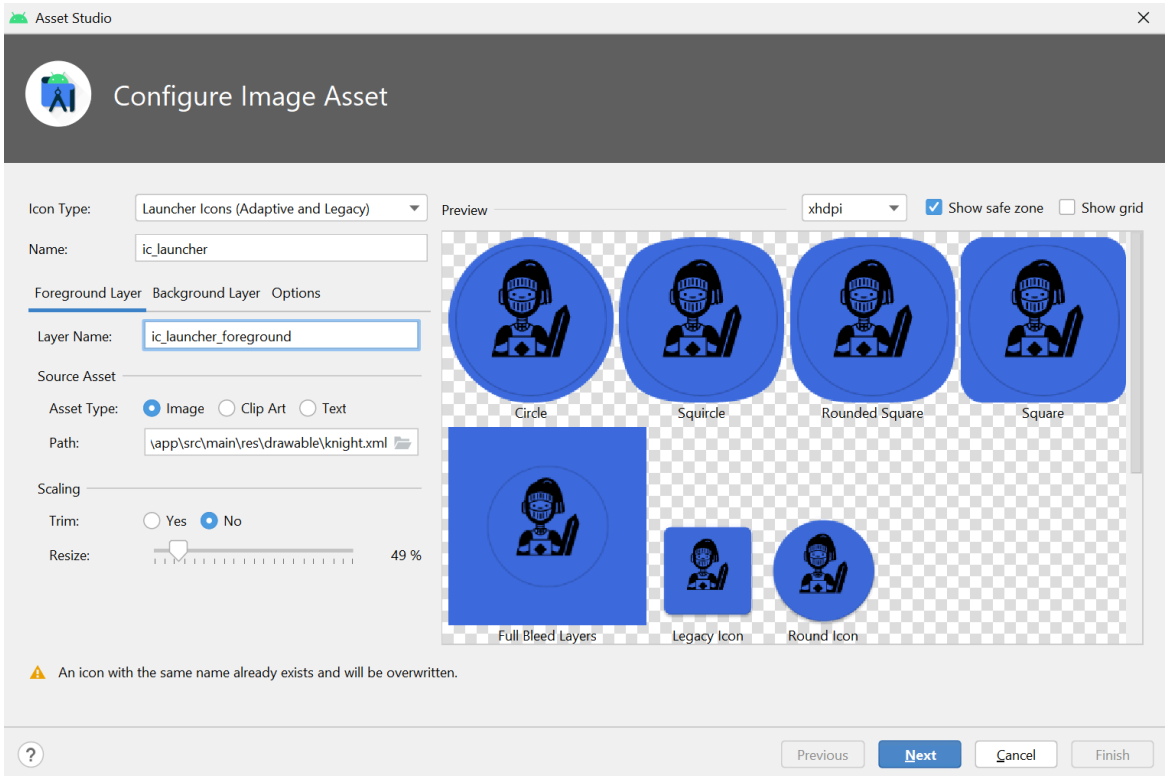
Smažte následující soubory



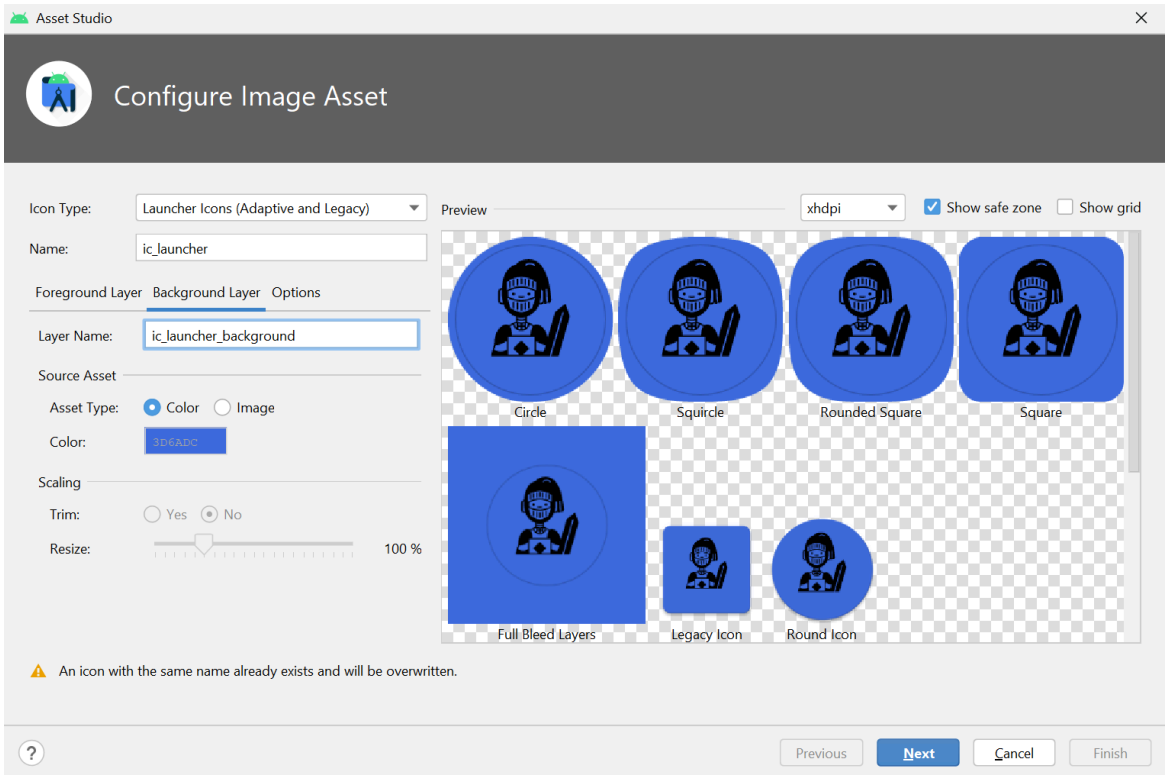
Vytvořte nový Image Asset.



U ikony se definuje popředí a pozadí. Vyberte nějaký obrázek do **Path** a případně upravte jeho velikost.



Na **Background Layer** upravte barvu. A proces vytváření ikony dokončete.

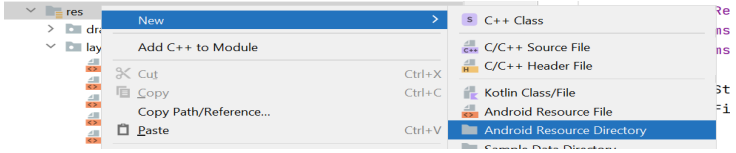


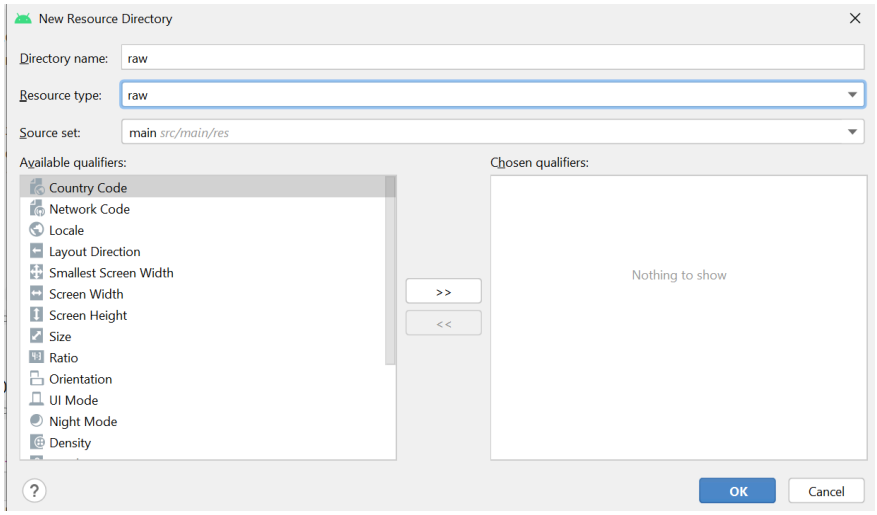
Načtení parametrů hry z JSON

Aktuálně kód obsahuje napevno definované prostředí hry (rozměr herní plochy, počet lesů, nepřátel, předměty, ...)

Toto nastavení přeneseneme do JSON souboru, což nám umožní vytvářet různé konfigurace her bez nutnosti měnit programový kód.

V resources si vytvoříme složku **raw**. Vytvoříme **Android Resource Directory** a typ nastavíme na **raw**.





Do složky vložíme **settings.json**, který stáhnete z tohoho kurzu. Soubor obsahuje nastavení hry, hrdiny, nepřátel a předmětů. Seznamte se s jeho strukturou.

Čtení souboru settings.json zajistíme metodou **readSettingsJson** v aktivitě GameActivity. Metoda otevře soubor z Resources/Raw, načte jeho řádky a vrátí ho jako string.

```
fun readSettingsJson () : String {
    var string: String? = ""
    val stringBuilder = StringBuilder()
    val inputStream: InputStream = this.resources.openRawResource(R.raw.settings)
    val reader = BufferedReader(InputStreamReader(inputStream))
    while (true) {
        try {
            if (reader.readLine().also { string = it } == null) break
        } catch (e: IOException) {
            e.printStackTrace()
        }
        stringBuilder.append(string).append("\n")

    }
    inputStream.close()
    return stringBuilder.toString()
}
```

Spuštění metody umístíme do metody onCreate před vytváření instance hry pomocí GameFactory.

```
val settingsJson = readSettingsJson()
```

Načtený JSON řetězec pošleme na zpracování do třídy **Game**. Do konstuktoru Game přidáme vstupní parametr

```
class Game(
    heroName: String,
    settingsJson: String ="",
    ) : ViewModel() {
```

Spolu s tím musíme upravit **GameFactory**, abychom mohli řetězec předat do třídy Game

```
class GameFactory (
    private val heroName : String,
    private val settingsJson : String
): ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(Game::class.java)) {
            return Game(heroName, settingsJson) as T
        }
        throw IllegalArgumentException("Unknown Viewmodel class")
    }
}
```

Nastavení hry

Vytvoříme si novou datovou třídu **Settings** v package viewmodel. V konstruktoru bude parametr settingsJSON typu string.

```
data class Settings (var settingsJson : String ="") {
```

Třída bude zpracovávat vstupn JSON text pomocí knihovny **org.json** a jeho strukturu uloží do atributů třídy. U základních parametrů hry je datový typ jasný.

Hrdina má svoji strukturu, proto použijeme **JSONObject**. Nepřátelé a předměty jsou pole objektů, proto používáme **JSONArray**.

```
    var width = 20
    var height = 10
    var numForests = 5
    var enemies : JSONArray?
    var hero: JSONObject?
```

Ve třídě **Settings** is vytvoříme metodu **parseJsonInt**, která z JSON řetězce extrahuje číselnou položku zadanou jménem. Protože si nejsme jisti chybovostí vstupního JSON souboru, ošetřujeme zpracování pomocí výjimek.

Nejprve se zavolá funkce **JSONObject**, která provede parsování JSON a vrátí JSONObject. Následně si z něj pomocí **getInt** přečteme hodnotu zadané položky item.

Pokud se něco nepovede pomocí Log vrátíme do logu chybovou hlášku. **e** je pro chybu. **i** pro info, **d** pro debug, **w** pro varování, apod.

```
fun parseJsonInt(settingsJson:String, item:String) : Int? {
    try {
        val jObj = JSONObject(settingsJson)
        return jObj.getInt(item)
    } catch (ex: JSONException) {
        Log.i("JsonParser Int", "unexpected JSON exception", ex)
        return null
    }
}
```

Do třídy přidáme init blok, kde načteme základní parametry hry. Pokud parsování nevrátí null, tak hodnotu uložíme do atributu třídy, jinak se použije výchozí hodnota

```
init {
    parseJsonInt(settingsJson, "width")?.let { width = it }
    parseJsonInt(settingsJson, "height")?.let { height = it }
    parseJsonInt(settingsJson, "numforests")?.let { numForests = it }
}
```

Vytvoříme velmi podobné funkce **parseJsonObject** a **parseJsonArray**, které použijeme na načtení hrdiny, nepřátel a předmětů.

```
fun parseJsonObject(settingsJson:String, item:String) : JSONObject? {
    try {
        val jObj = JSONObject(settingsJson)
        return jObj.getJSONObject(item)
    } catch (ex: JSONException) {
        Log.i("JsonParser object", "unexpected JSON exception", ex)
        return null
    }
}

fun parseJsonArray(settingsJson:String, item:String) : JSONArray? {
    try {
        val jObj = JSONObject(settingsJson)
        return jObj.getJSONArray(item)
    } catch (ex: JSONException) {
        Log.i("JsonParser array", "unexpected JSON exception", ex)
        return null
    }
}
```

Dokončíme init blok

```
init {
    parseJsonInt(settingsJson, "width")?.let { width = it }
    parseJsonInt(settingsJson, "height")?.let { height = it }
    parseJsonInt(settingsJson, "numforests")?.let { numForests = it }
    hero = parseJsonObject(settingsJson, "hero")
    enemies = parseJsonArray(settingsJson, "enemies")
}
```

Úprava generování herního plánu

Ve třídě Game nahradíme atributy width, height, numForrest, numEnemies za settings

```
private var settings = Settings(settingsJson)
```

Upravíme vytváření herního plánu

```
private var gamePlan = GamePlan (settings.width, settings.height, settings.numForests)
```

Úprava generování hrdiny

Změníme definici atributu hero a drobně upravíme init blok.

```
lateinit var hero : Hero
```

```
init {
    generateHero(settings.hero, heroName)
    generateEnemies(settings.enemies)

```

Vytvoříme si novou metodu generateHero, která bude mít na vstupu jméno hrdiny a JSONObject o hrdinovi.

Protože JSON nemusí hrdinu obsahovat, ponecháme i vytváření hrdiny s výchozími hodnotami.

Pro vytvoření hrdiny s parametry s JSON použijeme plný constructor.

```
private fun generateHero(heroSettings: JSONObject?, heroName: String) {
    if (heroSettings != null) {

        hero = Hero (name = heroName,
            position = gamePlan.generateRandomPositionOnMeadow(),
            health = heroSettings.getDouble("health"),
            attack = heroSettings.getDouble("attack"),
            defense = heroSettings.getDouble("defense"),
            healing = heroSettings.getDouble("healing")
        )
    } else {
        hero = Hero(name = heroName, position = gamePlan.generateRandomPositionOnMeadow())
    }

    gameObjects.add(hero)
}
```

Úprava generování nepřátel

Pro generování nepřátel upravíme metodu generateEnemies.

Opět musíme ošetřit možnost null hodnoty. U každého typu nepřítele je uveden parametr count, který určuje kolik nepřátel tohoto typy vygenerovat.

```
private fun generateEnemies(enemies: JSONArray?) {
    var enemy : Enemy
    enemies?.let {
        for (i in 0 until enemies.length()) {
            val enemyType = enemies.getJSONObject(i)
            repeat (enemyType.getInt("count")) {
                enemy = Enemy(name = enemyType.getString("name"),
                    position = gamePlan.generateFreeRandomPositionOnMeadow(gameObjects),
                    health = enemyType.getDouble("health"),
                    attack = enemyType.getDouble("attack"),
                    defense = enemyType.getDouble("defense"),
                )
                this.enemies.add(enemy)
                gameObjects.add(enemy)
            }
        }
    }
}
```

Git

Úpravy po vyzkoušení vložte do gitu.

Vytvořeno v rámci projektu: **DANTE**, reg. č. NPO_UPCE_MSMT-16591/2022

Financováno Evropskou unií
NextGenerationEU
Národní plán obnovy
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY

Jetpack Compose - úvod

Jetpack Compose je moderní sada nástrojů pro vytváření nativního uživatelského rozhraní systému Android. Jetpack Compose zjednodušuje a urychluje vývoj uživatelského rozhraní v systému Android díky menšímu množství kódu, výkonným nástrojům a intuitivnímu rozhraní Kotlin API.

V tomto kurzu vytvoříte jednoduchou komponentu uživatelského rozhraní s deklarativními funkcemi. Nebudete upravovat žádné rozvržení XML ani používat Editor rozvržení. Místo toho budete volat composable funkce, abyste definovali, jaké prvky chcete, a o zbytek se postará kompilátor Compose.

Composable funkce

Aplikace Jetpack Compose je postavena na kompozitních funkcích. Tyto funkce se v angličtině označují jako composable, a protože by se těžko překládaly, necháme je jako composable. Tyto funkce umožňují programově definovat uživatelské rozhraní aplikace tak, že popisují, jak má vypadat, a poskytují datové závislosti, místo aby se zaměřovaly na proces konstrukce uživatelského rozhraní (inicializace prvku, jeho připojení k nadřazenému prvku atd.). Chcete-li vytvořit komponovatelnou funkci, stačí k jejímu názvu přidat anotaci @Composable.

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.material3.Text

třída MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContent {
         Text("Hello world!")
      }
   }
}
```

Přidání textového prvku

Chcete-li začít, stáhněte si nejnovější verzi aplikace Android Studio, vytvořte aplikaci výběrem možnosti Nový projekt a v kategorii Telefon a tablet vyberte možnost Prázdná aktivita. Pojmenujte aplikaci ComposeTutorial a klepněte na tlačítko Finish. Výchozí šablona již obsahuje některé prvky Compose, ale v tomto návodu ji budete postupně vytvářet.

Nejprve napište text "Hello world!" přidáním textového prvku do metody onCreate. To provedete definováním bloku obsahu a voláním složené funkce Text. Blok setContent definuje rozvržení aktivity, ve kterém se volají kompozitní funkce. Kompozitní funkce lze volat pouze z jiných kompozitních funkcí.

Jetpack Compose používá zásuvný modul kompilátoru Kotlin k transformaci těchto composable funkcí na prvky uživatelského rozhraní aplikace. Například composable funkce Text, která je definována knihovnou Compose UI, zobrazí na obrazovce textový popisek.

Definice composable funkce

Chcete-li, aby byla funkce composable, přidejte anotaci @Composable. Chcete-li si to vyzkoušet, definujte funkci MessageCard, které je předáno jméno a která ho použije ke konfiguraci textového prvku.

Náhled funkce v aplikaci Android Studio

Anotace @Preview umožňuje zobrazit náhled composable funkcí v aplikaci Android Studio, aniž byste museli aplikaci sestavovat a instalovat do zařízení Android nebo emulátoru. Anotaci je třeba použít u kompozitní funkce, která nepřijímá parametry. Z tohoto důvodu nelze přímo zobrazit náhled funkce MessageCard. Místo toho vytvořte druhou funkci s názvem PreviewMessageCard, která zavolá funkci MessageCard s příslušným parametrem. Před @Composable přidejte anotaci @Preview.

Přestavte svůj projekt. Samotná aplikace se nezmění, protože nová funkce PreviewMessageCard není nikde volána, ale Android Studio přidá okno náhledu, které můžete rozbalit kliknutím na rozdělené zobrazení (návrh/kód). Toto okno zobrazuje náhled prvků uživatelského rozhraní vytvořených composable funkcemi označenými anotací @Preview. Chcete-li náhledy kdykoli aktualizovat, klepněte na tlačítko obnovit v horní části okna náhledu.

Rozložení

Prvky uživatelského rozhraní jsou hierarchické, prvky jsou obsaženy v jiných prvcích. V aplikaci Compose vytváříte hierarchii uživatelského rozhraní voláním composable funkcí z jiných composable funkcí.

Přidání více textů

Zatím jste vytvořili svou první kompozitní funkci a náhled! Abyste objevili další možnosti Jetpack Compose, vytvoříte jednoduchou obrazovku pro zasílání zpráv obsahující seznam zpráv, který lze rozšířit o několik animací.

Začněte tím, že zprávu obohatíte o jméno jejího autora a obsah zprávy. Nejprve je třeba změnit parametr composable tak, aby přijímal objekt Message namísto řetězce String, a přidat další composable Text uvnitř composable MessageCard. Nezapomeňte také aktualizovat náhled.

```
třída MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      MessageCard(Zpráva("Android", "Jetpack Compose"))
    }
```

```
    }
}

data class Message(val author: String, val body: String)

@Composable
fun MessageCard(msg: Message) {
    Text(text = msg.author)
    Text(text = msg.body)
}

@Preview
@Composable
fun PreviewMessageCard() {
    MessageCard(
        msg = Message("Lexi", "Hey, take a look at Jetpack Compose, it's great!")
    )
}
```

Tento kód vytvoří dva textové prvky uvnitř zobrazení obsahu. Protože jste však neposkytli žádné informace o tom, jak je uspořádat, textové prvky se vykreslí nad sebou, takže text je nečitelný.

Použití sloupce

Na stránkách Sloupec umožňuje uspořádat prvky vertikálně. Přidejte funkci Sloupec do funkce MessageCard.
Můžete použít Řádek k vodorovnému uspořádání prvků a Box k uspořádání prvků na sebe.

Přidání prvku obrázku

Obohaťte svou kartu se zprávou přidáním profilové fotografie odesílatele. Použijte zdrojůSprávce importovat obrázek z knihovny fotografií nebo použijte nástroj this tento.
Přidejte kompozitní řádek, abyste měli dobře strukturovaný design, a uvnitř něj kompozitní obrázek.

Konfigurace rozložení

Rozložení vaší zprávy má správnou strukturu, ale její prvky nejsou dobře rozmístěny a obrázek je příliš velký! K ozdobení nebo konfiguraci kompozitního souboru používá nástroj Compose **modifikátory**. Ty umožňují měnit velikost, rozložení, vzhled kompozitního prvku nebo přidávat interakce na vysoké úrovni, jako je například možnost kliknutí na prvek. Můžete je řetězit a vytvářet tak bohatší kompozitní soubory. Některé z nich použijete k vylepšení rozvržení.

Material Design

Aplikace Compose je vytvořena s podporou principů Material Designu. Mnoho prvků uživatelského rozhraní implementuje Material Design hned po vybalení. V této lekci budete stylizovat svou aplikaci pomocí widgetů Material Designu.

Použití Material Designu

Návrh vaší zprávy má nyní rozvržení, ale zatím nevypadá skvěle.

Jetpack Compose poskytuje implementaci Material Designu 3 a jeho prvků uživatelského rozhraní hned po vybalení. Vylepšíte vzhled naší kompozitní karty MessageCard pomocí stylů Material Designu.

Na začátku obalte funkci MessageCard motivem Material vytvořeným ve vašem projektu, ComposeTutorialTheme, a také motivem Surface. Udělejte to jak ve funkci @Preview, tak ve funkci setContent. Díky tomu budou vaše kompozitní objekty dědit styly definované v tématu vaší aplikace, což zajistí konzistenci celé aplikace.

Material Design je postaven na třech pilířích: Barva, typografie a tvar. Budete je přidávat jeden po druhém.

```
třída MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContent {
         ComposeTutorialTheme {
            Surface(modifier = Modifier.fillMaxSize()) {
               MessageCard(Zpráva("Android", "Jetpack Compose"))
            }
         }
      }
   }
}

@Preview
@Composable
fun PreviewMessageCard() {
   ComposeTutorialTheme {
      Povrch {
         MessageCard(
            msg = Message("Lexi", "Podívejte se na Jetpack Compose, je skvělý!")
         )
      }
   }
}
```

Barva

Pomocí MaterialTheme.colorScheme můžete stylovat pomocí barev ze zabaleného motivu. Tyto hodnoty z motivu můžete použít kdekoli, kde je potřeba barva. Tento příklad používá dynamické barvy tématu (definované předvolbami zařízení). Chcete-li to změnit, můžete v souboru MaterialTheme.kt nastavit hodnotu dynamicColor na false.

Stylujte nadpis a přidejte k obrázku rámeček.

Typografie

Styly Material Typography jsou k dispozici v MaterialTheme, stačí je přidat do složek Text.

Tvar

Pomocí aplikace Shapeyou můžete přidat poslední úpravy. Nejdříve obtočte text těla zprávy kolem písmene Surface kompozitním. To umožňuje přizpůsobit tvar a výšku těla zprávy. Ke zprávě se také přidá polstrování pro lepší rozvržení.

Povolení tmavého motivu

Tmavé téma (nebo noční režim) lze zapnout, abyste se vyhnuli jasnému displeji zejména v noci, nebo abyste jednoduše šetřili baterii zařízení. Díky podpoře Material Designu zvládá Jetpack Compose tmavé téma ve výchozím nastavení. Po použití barev Material Designu se text a pozadí automaticky přizpůsobí tmavému pozadí.

V souboru můžete vytvořit více náhledů jako samostatné funkce nebo do jedné funkce přidat více poznámek.

Přidejte novou anotaci náhledu a povolte noční režim.
Volby barev pro světlé a tmavé motivy jsou definovány v souboru Theme.kt vygenerovaném prostředím IDE.

Zatím jste vytvořili prvek uživatelského rozhraní pro zprávy, který zobrazuje obrázek a dva texty s různými styly a vypadá dobře jak ve světlých, tak v tmavých tématech!

Seznamy a animace

Seznamy a animace jsou v aplikacích všude. V této lekci se dozvíte, jak Compose umožňuje snadné vytváření seznamů a zábavné přidávání animací.

Vytvoření seznamu zpráv

Konverzace s jednou zprávou působí trochu osaměle, a proto se chystáme změnit konverzaci tak, aby obsahovala více než jednu zprávu. Budete muset vytvořit funkci Konverzace, která bude zobrazovat více zpráv. Pro tento případ použití použijte funkci Compose LazyColumn . a LazyRow .. Tyto kompozitní prvky vykreslují pouze prvky, které jsou viditelné na obrazovce, takže jsou navrženy tak, aby byly velmi efektivní pro dlouhé seznamy.

V tomto úryvku kódu vidíte, že LazyColumn má potomka items. Jako parametr přijímá List a jeho lambda přijímá parametr, který jsme pojmenovali message (mohli jsme ho pojmenovat jakkoli), což je instance Message. Stručně řečeno, tato lambda je volána pro každou položku poskytnutého Seznamu. Zkopírujte ukázkovou datovou sadu do svého projektu, abyste mohli rychle zavést konverzaci.

```
// …
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items

@Composable
fun Conversation(messages: List<Message>) {
   LazyColumn {
      items(zprávy) { zpráva ->
         MessageCard(zpráva)
      }
   }
}

@Preview
@Composable
fun PreviewConversation() {
   ComposeTutorialTheme {
      Conversation(SampleData.conversationSample)
   }
}
```

Animovat zprávy při rozšiřování

Konverzace je stále zajímavější. Je čas pohrát si s animacemi! Přidáte možnost rozbalit zprávu a zobrazit delší zprávu, přičemž se animuje velikost obsahu i barva pozadí. Abyste mohli tento lokální stav uživatelského rozhraní ukládat, musíte sledovat, zda byla zpráva rozbalena, nebo ne. Pro sledování této změny stavu musíte použít funkce remember a mutableStateOf.

Kompozitní funkce mohou ukládat lokální stav do paměti pomocí funkce remember a sledovat změny hodnoty předané funkci mutableStateOf. Kompozitní funkce (a jejich potomci) používající tento stav se při aktualizaci hodnoty automaticky překreslí. To se nazývá rekompozice.

Pomocí stavových rozhraní API Compose, jako jsou remember a mutableStateOf, se při jakékoli změně stavu automaticky aktualizuje uživatelské rozhraní.

Nyní můžete změnit pozadí obsahu zprávy na základě isExpanded, když klikneme na zprávu. K obsluze událostí kliknutí na kompozitum použijete modifikátor clickable. Namísto pouhého přepínání barvy pozadí plochy budete barvu pozadí animovat postupnou změnou její hodnoty z MaterialTheme.colorScheme.surface na MaterialTheme.colorScheme.primary a naopak. K tomu použijete funkci animateColorAsState. Nakonec použijete modifikátor animateContentSize pro plynulou animaci velikosti kontejneru zprávy.

Životní cyklus funkce composable

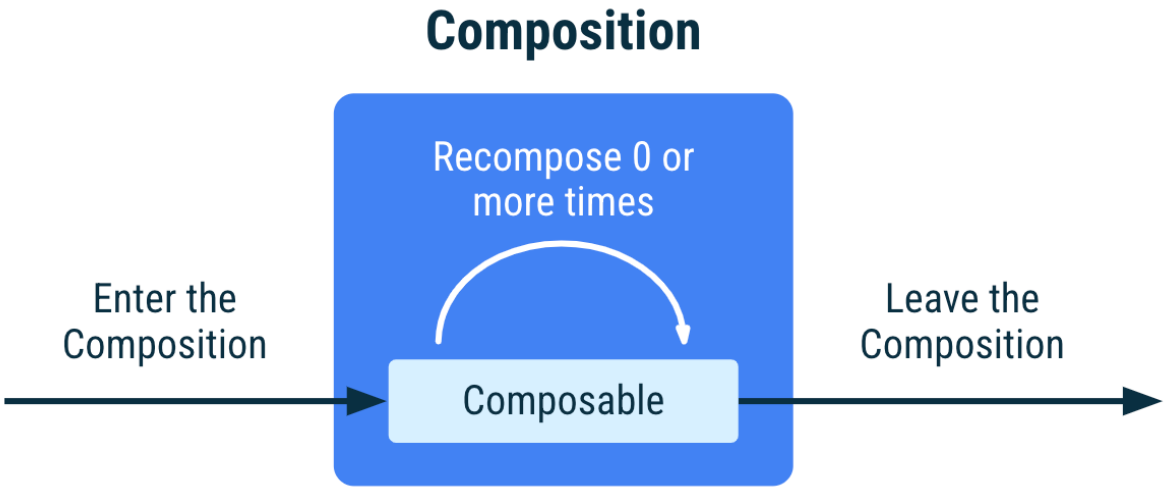
Na této stránce se dozvíte o životním cyklu kompozitního souboru a o tom, jak Compose rozhoduje o tom, zda kompozitní soubor potřebuje rekompozici.

Přehled životního cyklu

Jak je uvedeno v Správa státní dokumentace, kompozice popisuje uživatelské rozhraní vaší aplikace a vzniká spuštěním kompozitních souborů. Kompozice je stromová struktura composable objektů, které popisují vaše uživatelské rozhraní.

Když Jetpack Compose poprvé spustí vaše composable objekty, během *počátečního skládání* bude sledovat composable objekty, které jste zavolali k popisu uživatelského rozhraní v kompozici. Když se pak stav vaší aplikace změní, naplánuje Jetpack Compose *opětovné složení*. Rekompozice spočívá v tom, že Jetpack Compose znovu provede kompozitní objekty, které se mohly změnit v reakci na změny stavu, a poté aktualizuje Kompozici tak, aby odrážela všechny změny.

Kompozice může být vytvořena pouze počáteční kompozicí a aktualizována rekompozicí. Jediný způsob, jak upravit kompozici, je rekompozice.



Obrázek 1 - Životní cyklus kompozitního prvku v kompozici. Vstoupí do Kompozice, 0krát nebo vícekrát se rekomponuje a opustí Kompozici.

Rekompozice se obvykle spouští změnou StateState<T> objektu. Compose je sleduje a spouští všechny kompozitní objekty v Composition, které čtou daný State<T>, a všechny jimi volané kompozitní objekty, které nelze přeskočit.

Pokud je komponovatelný objekt volán vícekrát, je do kompozice umístěno více instancí. Každé volání má v Composition svůj vlastní životní cyklus.

```
@Composable
fun MyComposable() {
   Sloupec {
      Text("Hello")
      Text("World")
   }
}
```

Anatomie složeného textu v nástroji Composition

Instance composable v Composition je identifikována **místem volání**. Kompilátor Compose považuje každé místo volání za odlišné. Volání composable objektů z více míst volání vytvoří více instancí composable objektu v Composition.

Pokud během rekompozice volá složený objekt jiné složené objekty než při předchozí kompozici, Compose **zjistí, které složené objekty byly volány a které ne,** a u složených objektů, které byly volány v obou kompozicích, **se vyhne jejich rekompozici, pokud se jejich vstupy nezměnily**.

Zachování identity je klíčové pro přiřazení vedlejších efektů k jejich složitelnosti, aby se mohly úspěšně dokončit a ne restartovat při každé rekompozici.

Vezměme si následující příklad:

```
@Composable
fun LoginScreen(showError: Boolean) {
   if (showError) {
      LoginError()
   }
   LoginInput() // Toto volání stránky ovlivňuje, kde je LoginInput umístěn v Composition
}

@Composable
fun LoginInput() { /* ... */ }

@Composable
fun LoginError() { /* ... */ }
```

Ve výše uvedeném úryvku kódu bude LoginScreen podmíněně volat komponentu LoginError a vždy bude volat komponentu LoginInput. Každé volání má jedinečné místo volání a pozici zdroje, které kompilátor použije k jeho jednoznačné identifikaci.

Obrázek 2 - Zobrazení obrazovky LoginScreen v kompozici při změně stavu a rekompozici. Stejná barva znamená, že nedošlo k rekompozici.

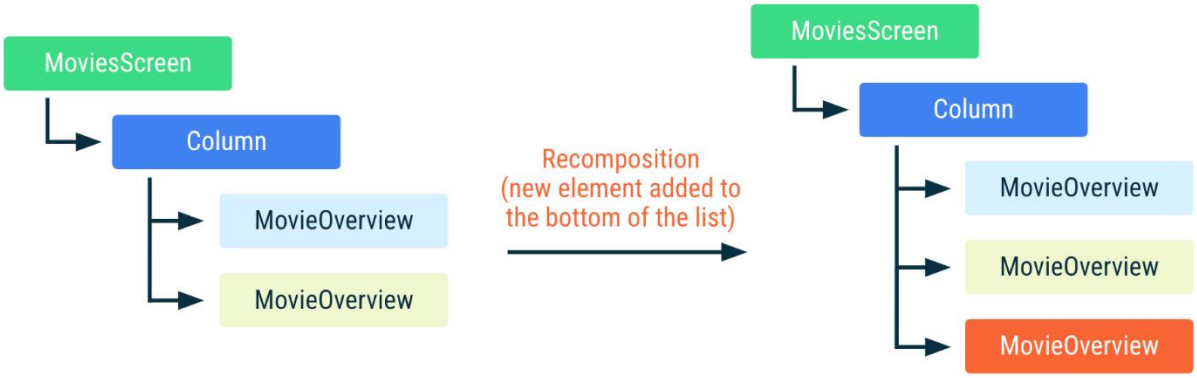
Přestože se funkce LoginInput začala volat jako první a nyní se volá jako druhá, instance LoginInput zůstane zachována napříč rekompozicemi. Navíc, protože LoginInput nemá žádné parametry, které by se napříč rekompozicemi změnily, volání LoginInput bude Compose přeskočeno.

Přidání dalších informací pro inteligentní rekompozice

Vícenásobné volání složené položky ji také vícenásobně přidá do složení. Při vícenásobném volání kompozitního prvku ze stejného místa volání nemá Compose žádné informace k jednoznačné identifikaci každého volání tohoto kompozitního prvku, takže se kromě místa volání používá i pořadí provádění, aby se instance rozlišovaly. Toto chování je někdy vše, co je potřeba, ale v některých případech může způsobit nežádoucí chování.

```
@Composable
fun MoviesScreen(movies: List<Movie>) {
   Sloupec {
      for (movie in movies) {
         // MovieOverview composable jsou umístěny v Composition daném jeho
         // indexovou pozici v cyklu for
         MovieOverview(movie)
      }
   }
}
```

Ve výše uvedeném příkladu Compose používá kromě místa volání také pořadí provádění, aby instance zůstala v Composition odlišná. Pokud je na *konec* seznamu přidán nový film, Compose může znovu použít instance, které již v Kompozici jsou, protože jejich umístění v seznamu se nezměnilo, a proto je vstup filmu pro tyto instance stejný.



Obrázek 3 - Zobrazení prvku MoviesScreen v kompozici při přidání nového prvku do spodní části seznamu. Kompozitní prvky MovieOverview v Kompozici lze použít opakovaně. Stejná barva v MovieOverview znamená, že kompozitum nebylo znovu složeno.

Pokud se však seznam filmů změní přidáním na *začátek* nebo *doprostřed* seznamu, odebráním nebo změnou pořadí položek, dojde k rekompozici ve všech voláních MovieOverview, jejichž vstupní parametr změnil pozici v seznamu. To je nesmírně důležité, pokud například MovieOverview načte obrázek filmu pomocí vedlejšího efektu. Pokud dojde k rekompozici v době, kdy efekt probíhá, bude zrušen a začne znovu.

```
@Composable
fun MovieOverview(movie: Movie) {
  Sloupec {
    // Vedlejší efekt vysvětlený později v dokumentaci. If MovieOverview
    // překomponuje, zatímco probíhá načítání obrázku,
    // je zrušeno a znovu spuštěno.
    val image = loadNetworkImage(movie.url)
    MovieHeader(image)

    /* ... */
  }
}
```

V ideálním případě bychom chtěli identitu instance MovieOverview považovat za spojenou s identitou filmu, který je jí předán. Pokud změníme pořadí seznamu filmů, v ideálním případě bychom podobně změnili pořadí instancí ve stromu Kompozice, místo abychom znovu skládali každý MovieOverview composable s jinou instancí filmu. Compose poskytuje způsob, jak běhovému prostředí sdělit, jaké hodnoty chcete použít k identifikaci dané části stromu: tzv. klíč composable.

Obalením bloku kódu voláním klíče composable s jednou nebo více předanými hodnotami se tyto hodnoty zkombinují a použijí se k identifikaci dané instance v kompozici. Hodnota klíče nemusí být *globálně* jedinečná, musí být jedinečná pouze mezi voláními kompozit na místě volání. V tomto příkladu tedy každý film musí mít klíč, který je jedinečný mezi filmy; nevadí, pokud tento klíč sdílí s nějakou jinou kompozicí jinde v aplikaci.

```
@Composable
fun MoviesScreenWithKey(movies: List<Movie>) {
  Sloupec {
    for (movie in movies) {
```

```
        key(movie.id) { // Unikátní ID pro tento film
          MovieOverview(movie)
        }
      }
    }
}
```

Pokud se prvky v seznamu změní, Compose rozpozná jednotlivá volání MovieOverview a může je znovu použít.

Některé composable soubory mají zabudovanou podporu pro composable klíč. Například LazyColumn akceptuje zadání vlastního klíče v DSL items.

```
@Composable
fun MoviesScreenLazy(movies: List<Movie>) {
  LazyColumn {
    items(movies, key = { movie -> movie.id }) { movie ->
      MovieOverview(movie)
    }
  }
}
```

Mobilní aplikace

Tabulka nejlepších her (SQLite, Room) - text

Tabulka nejlepších her

Pokud hráč dohraje úspěšně hru, uložíme jeho jméno a skóro do tabulky.

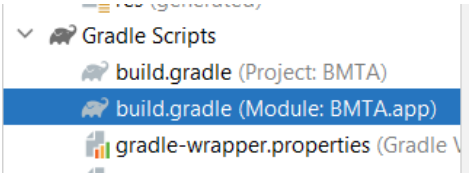
V rámci cvičení připravíme i aktivitu, která pomocí Recycler view zobrazí seřazené výsledky.

Pro uložení výsledků budeme používat jednoduchou SQLite databázi. Data budou uložena lokálně. Nebudeme provádět nějakou synchronizaci proti centrální databázi.

Konfigurace gradle

K databázi SQLite nebudeme přistupovat napřímo, ale použijeme knihovnu room. Knihovna zajišťuje trvalé uložení objektů do relační databáze. Řeší tedy mapovaní objektového modelu na relační model.

Abychom mohli knihovnu room používat musíme připravit kompilační prostředí. Otevřeme soubor build.gradle, do kterého jsme již přidávali podporu view binding.



Do pluginu přidáme kotlin-kapt, který zapíná podporu anotační procesoru.

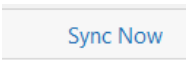
```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'kotlin-kapt'
}
```

Do závislostí přidáme anotační kompilátor, room runtime, podporu coroutines pro room a podporu coroutine pro lifecycle.

```
dependencies {
```

```
kapt 'androidx.room:room-compiler:2.4.3'
implementation 'androidx.room:room-runtime:2.4.3'
implementation 'androidx.room:room-ktx:2.4.3' // Kotlin Extensions and Coroutines support for Room
implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.5.1' // LifecycleScope
}
```

Nezapomeneme změny synchronizovat do projektu.



Entita

Pro přehlednost si vytvoříme package **database**. V něm budeme vytvářet potřebné třídy.

Prvním krokem je vytvoření třídy, která bude představovat ukládanou entitu. V našem případě to bude záznam úspěšné hry.

V package database vytvoříme datovou třídu **Score**. Atributy bude mít následující:

- **id** - číslo, které nám bude sloužit jako primární klíč a bude se automaticky generovat
- **dateAdded** - timestamp uložení výsledku. Použijem Java typ Date, který ukládá čas v podobě počtu milisekund od začátku epochy
- **name** - jméno hrdiny
- **score** - počet odehraných tahů

```
data class Score (
    val id : Int = 0,
    val dateAdded : Date,
    val name : String,
    val score : Int,
)
```

Nyní třídu označíme za entitu a začneme přidávat informace, které pomohou androidx.room s mapování objektu na relační tabulku. Stačí třídu anotovat jako Entitu.

```
@Entity
data class Score (
```

V tomto případě se vytvoří tabulka Score se sloupci pojmenovanými podle atributů konstruktoru.

Máme možnost, jak toto mapování zpřesnit.

- Sloupec můžeme označit za primární klíč. Pokud bychom chtěli mít klíč složený z více sloupců, tak ho zadefinujeme v rámci Entity
- Pokud před atribut dáme anotaci @ColumnInfo můžeme přesněji definovat sloupec v tabulce. například změnit jméno.
- Pokud nechceme nějaký sloupec do tabulky ukládat, můžeme před něj napsat @Ignore
- V entitě můžeme definovat jméno tabulky, indexy. Případně složený primární klíč, více ignorovaných sloupců apod.

```
@Entity (
    tableName = "score",
    indices = arrayOf(Index(value = arrayOf ("score"), unique = false),
                      Index(value = arrayOf("name", "score")))
    // primaryKeys = arrayOf ("id", "dateAdded"),
    // ignoredColumns = arrayOf ("ignore", "test")
        )
data class Score (
    @PrimaryKey(autoGenerate = true) val id : Int = 0,
    @ColumnInfo(name="dateAdded") val date : Date,
    val name : String,
    val score : Int
) {
//@Ignore val ignore : String
}
```

DAO

Dalším krokem je definice rozhraní, přes který budeme s Entitou komunikovat. Obsahuje metody pro čtení, vkládání, aktualizaci a mazání dat.

Vytvoříme nový interface **ScoreDAO**

```
interface ScoreDao {}
```

Interface označíme jako **DAO** (Data Access Object)

```
@Dao
interface ScoreDao {}
```

Postupně budeme přidávat metody, které budou realizovat operace INSERT, UPDATE, DELETE, SELECT. Metoda musí být označena správnou anotací. Anotace může obsahovat detailnější chování operace.

Metod pro insert může být více, ale parametry musí být typu Entita. Buď může být 1, nebo n. Lze zadat List<Score> nebo použít varagr. **Pokud metoda obsahuje pouze jednu entitu může vrátit hodnotu Long, ve které bude vygenerované ID.**

```
@Insert (onConflict = ABORT)
fun addScore (score: Score) : Long
```

```
@Insert
fun addScores (vararg scores: Score)
```

Metody pro aktualizaci vypadají velmi podobně. Pouze se anotují pomocí **@Update** nebo **@Delete**. Řádek v tabulce se vyhledá podle primárního klíče a podle obsahu objektu se aktualizuje.

```
@Delete
fun deleteScore (score: Score) : Int
```

Metody pro získání dat začínají **@Query** a v závorce je SQL dotaz. Výstupem je seznam entit. Protože nám databáze běží na pozadí, obalí se List do Flow. **Flow** je interface v **coroutines** a řeší přenosy mezi vlákny.

Pokud by se měly vracet pouze některé sloupce, je potřeba vytvořit pomocnou Entitu, která bude obsahovat pouze některé sloupce.

```
@Query ("SELECT * FROM score ORDER BY score")
fun getAllScores(): Flow<List<Score>>
```

Lze definovat i parametrické dotazy, kdy parametr metody se promítne do SQL.

```
@Query ("SELECT * FROM score where name = :name")
fun getPlayerScore (name : String) : Flow<List<Score>>
```

Pokud potřebujeme napsat příkaz, který hromadně modifikuje data, označíme ho jako Query.

```
@Query ("DELETE FROM score where name = :name")
fun deletePlayerScore (name : String)
```

Databáze

Dále budeme potřebovat třídu pro databázi. Vytvoříme si abstraktní třídu **ScoreDatabase**, která bude dědit z RoomDatabase.

```
abstract class ScoreDatabase : RoomDatabase() {
}
```

Třídu anotujeme jako **Database**. Některá parametry jsou povinné - seznam Entit a verze databáze. Verzí databáze se rozumí verze datového modelu. Pokud budeme aplikaci rozšiřovat a bude se měnit datový model, musí se vyřešit, jak aplikace má pracovat s existující původní verzí databáze. Při změně je do aplikace dopsat migrační metody, které například upraví tabulky při přechodu z verze 1 na verzi 2.

```
@Database (
    entities = [Score::class],
    version = 1
        )
abstract class ScoreDatabase : RoomDatabase() {
}
```

Do třídy přidáme funkci, která vrátí DAO pro ovládání tabulek. Pokud bychom měli více entit, budeme mít více DAO.

```
abstract class ScoreDatabase : RoomDatabase() {
    abstract fun scoreDao(): ScoreDao
}
```

Součástí třídy bude i **companion object.** Jedná se o ekvivalent **statické metody třídy**. Metody mohu volat, aniž bych měl instanci třídy database.

Companion object obsahuje interní proměnnou INSTANCE, ve které budeme uchovávat odkaz na třídu databáze.

Metoda **getDatabase** vrátí nebo vytvoří datazi. Pro vytvoření databáze se používá metoda **buildDatabase**.

Metoda buildDatabase používá aplikační kontext, třídu databáze a jméno databáze.

```
companion object {
    @Volatile
    private var INSTANCE: ScoreDatabase? = null

    fun getDatabase(context: Context): ScoreDatabase {
        if (INSTANCE == null) {
            synchronized(this) {
                INSTANCE = buildDatabase(context)
            }
        }
        return INSTANCE!!
    }

    private fun buildDatabase(context: Context): ScoreDatabase {
        return Room.databaseBuilder(
            context.applicationContext,
            ScoreDatabase::class.java,
            "score_database"
        )
            .build()
    }
}
```

Při používání databází se používá vícevláknová architektura. V hlavním vlákně běží aplikace a dalším vlákně běží databáze. Databázové operace se provádí samostatně a v některých případech asynchronně. Aplikace chce uložit data, předá je databázi běžící v jiném vláknu k uložení. Aplikace pokračuje v další činnost a k uložení dat dojde napozadí.

Některé činnosti musí být synchronizované, proto vytváření databáze voláme v synchronizačním bloku.

Datová třída Score obsahuje atribut data typu **Date**, který SQLite nezná. Potřebujeme vytvořit typový konventor, který typ date převede do typu známého SQLite.

Vytvoříme třídu **ScoreConverters**. Do ní umístíme metody fromTimestamp a dateToTimeStamp.

```
class ScoreConverters {
    @TypeConverter
    fun fromTimestamp(value: Long?): Date? {
        return value?.let { Date(it) }
    }

    @TypeConverter
    fun dateToTimestamp(date: Date?): Long? {
        return date?.time
    }
}
```

A třídě ScoreDatabase mimo anotace Database přidáme další anotaci TypeConverters.

```
@TypeConverters(ScoreConverters::class)
abstract class ScoreDatabase : RoomDatabase() {
```

Zapojení databáze do aplikace

Pokud hra skončí zabitím všech nepřátel, vytvoříme záznam a uložíme ho do databáze.

Pro manipulaci s databází potřebujeme do viewmodelu Game dostat DAO.

Ve třídě **Game** do konstruktoru přidáme parametr **scoreDao**

```
class Game(
    heroName: String,
    settingsJson: String ="",
    val scoreDao: ScoreDao
    ) : ViewModel() {
```

Stejně tak budeme muset upravit třídu **GameFactory**, která nám vytváří instanci třídy Game.

```
class GameFactory (
    private val heroName : String,
    private val settingsJson : String,
    private val scoreDao : ScoreDao
): ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(Game::class.java)) {
            return Game(heroName, settingsJson, scoreDao) as T
        }
        throw IllegalArgumentException("Unknown Viewmodel class")
    }
}
```

Do aktivity **GameActivity** přidáme atribut **scoreDao**. lazy operátor určuje, že se atribut naplní, až ve chvíli, kdy bude potřeba. Protože třída ScoreDatabase obsahuje companion objekt, nemusíme vytvářet její instanci a pouze zavoláme metody na vytvoření databáze a vrácení Dao.

```
private val scoreDao by lazy { ScoreDatabase.getDatabase(this).scoreDao() }
```

Získaný DAO předáme přes Factory.

```
val factory = GameFactory(heroName, settingsJson, scoreDao)
```

Ve třídě **Game** upravíme metodu **isGameFinishing**, aby vytvořila skoré s aktuálním datumem a to přes DAO uložila do databáze. Protože se používá Coroutines musí se vložení spustit pomocí **GlobalScope**.

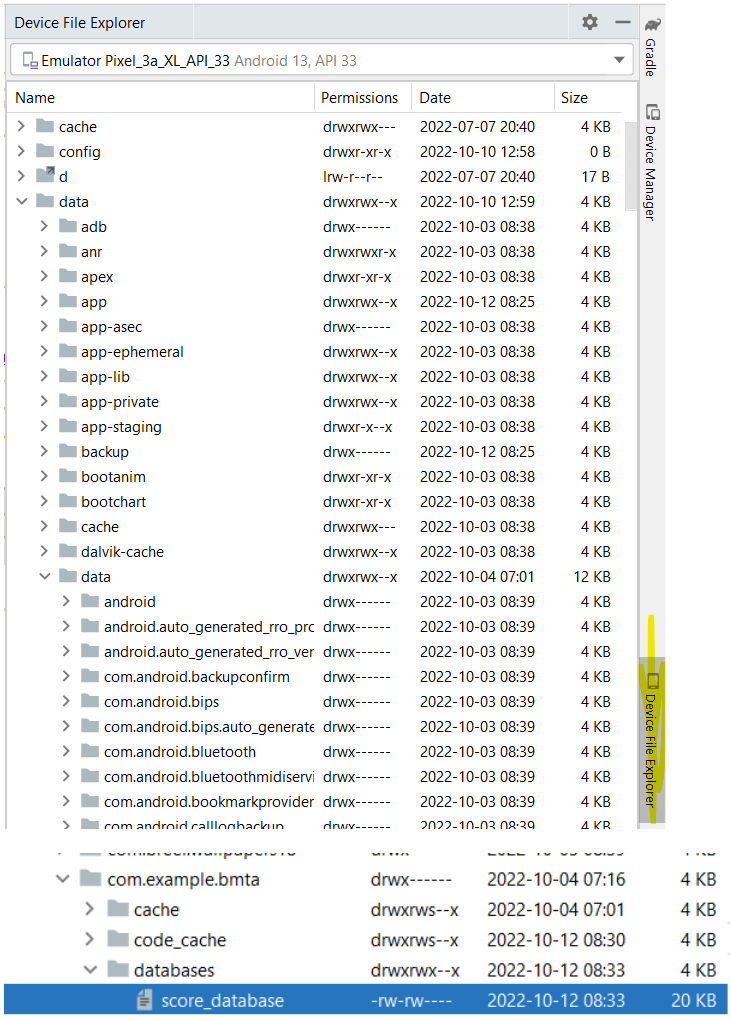
```
var scoreSaved = false
```

```
fun isGameFinish(): String {
    if (hero.isDeath()) return "Jsi mrtvý."
    if (allEnemiesDead()) {
        if (! scoreSaved) {
            GlobalScope.launch {
                scoreDao.addScore(Score(0, Date(), hero.name, score))
            }
            scoreSaved = true
        }
        return "Všichni nepřátelé jsou mrtví. Vyhrál jsi. Potřeboval jsi $score tahů."
    }
    if (command == "konec") return "Konec hry."
    return ""
}
```

Ověření ukládání

Odehrajeme úspěšně hru.

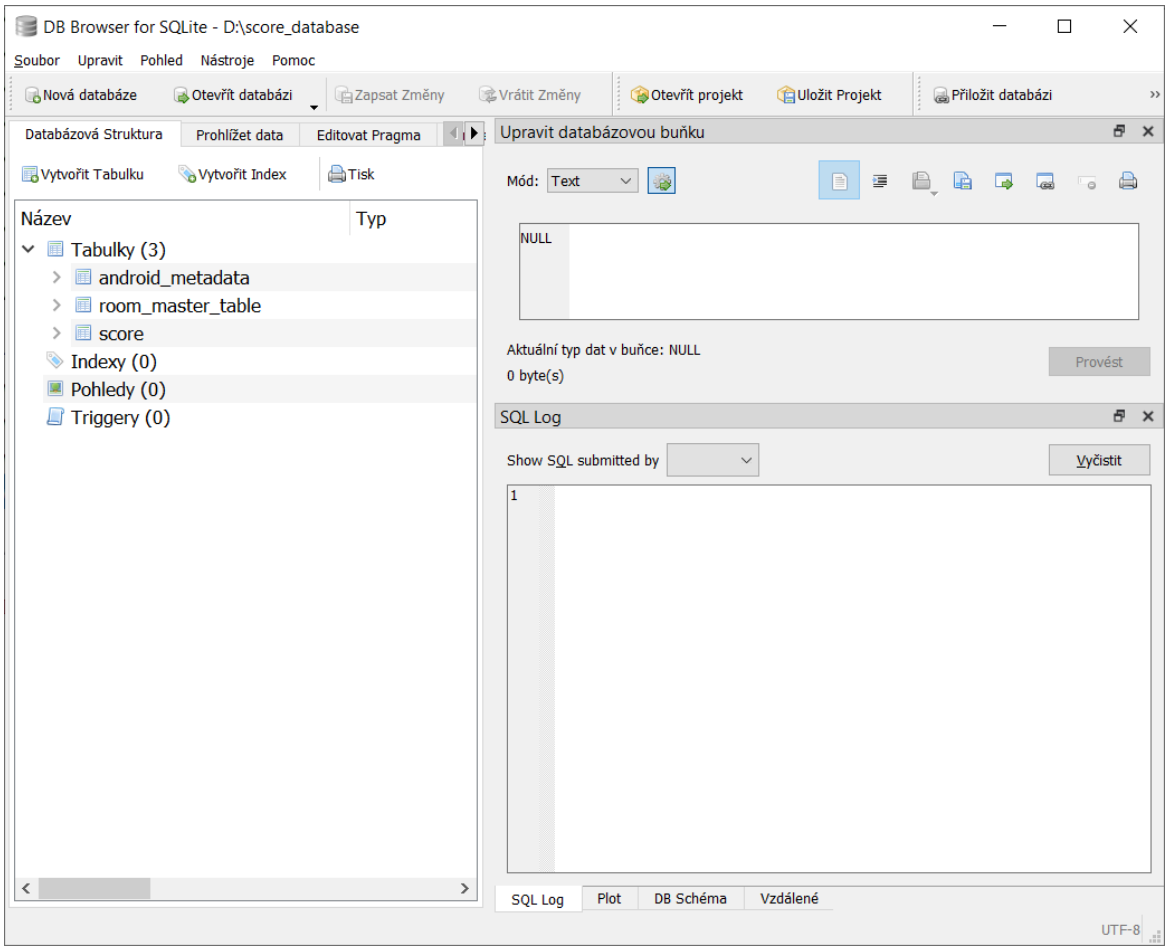
Spustíme si Device File Explorer. A budeme se navigovat do /data/data/jméno package/database.

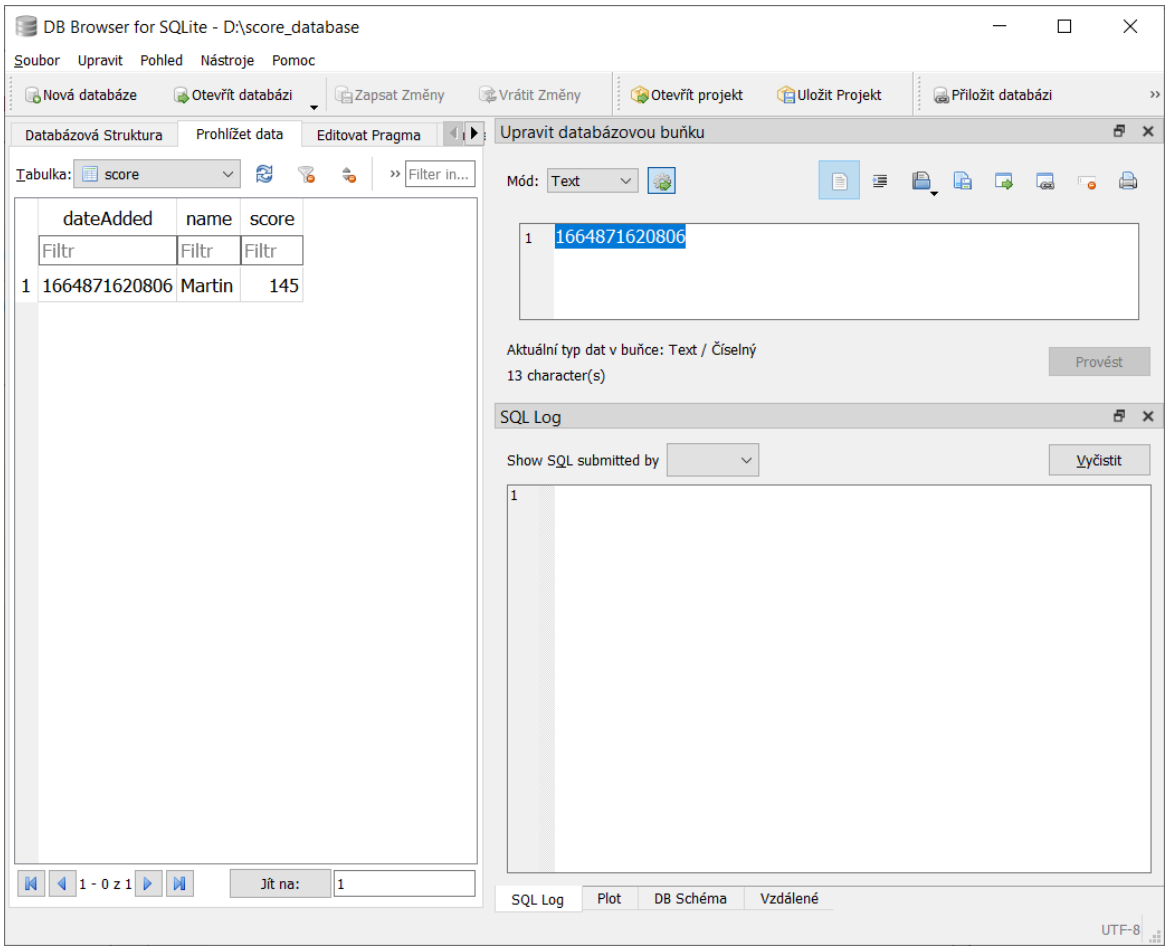


Označíme soubor score_database v kontextovém menu použijeme Save as. Databázi si uložíme na lokální disk.

Pro zobrazení obsahu dat můžeme použít například DB Browser for SQLite.

Měli bychom vidět, že vytvořili 3 tabulky. A v záložce Prohlížet data se můžeme podívat, že se data opravdu uložila.





Zobrazení výsledků

Vytvoříme constraint layout **score_row**. Přidáme dva textView, správně je ukotvíme, nastavíme styly apod.

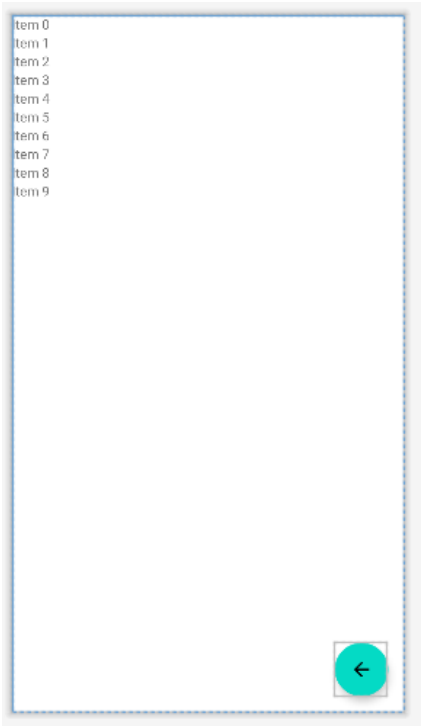
- **name**

- **score**

U samotného layoutu nezapomene nastavit **layout_height** na **wrap_content**. V opačném případě by nám pak každý záznam zabíral celou obrazovku.

Vytvoříme novou prázdnou aktivitu **ScoreActivity**. Do ní vložíme Recycler view s id **scoreRecyclerView**. Recycler view ukotvíme na celou obrazovku.

Přidáme **Floating Action Button**.Jako obrázek můžeme použít například **homeAsUpIndicator**. Id mu nastavíme na **backBtn**. Tlačítko ukotvíme.



Z hlavní aktivitě **MainActivity** po kliknutí na tlačítko Výsledky vyvoláme aktivitu ScoreActivity.

```
binding.resultsBtn.setOnClickListener() {
    startActivity(Intent(this, ScoreActivity::class.java))
}
```

Upravíme **ScoreActivity**, aby používala View Binding. Při kliknutí na tlačítko zpět se aktivita ukončí.

```
class ScoreActivity : AppCompatActivity() {
    private lateinit var binding : ActivityScoreBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityScoreBinding.inflate(layoutInflater)
        setContentView(binding.root)

        binding.backBtn.setOnClickListener() {
            finish()
        }
    }
}
```

Pro Recycler view potřebujeme Adapter a Holder. Tentokrát nepoužijeme RecyclerView.Adapter, ale jeho **ListAdapter**, který je jeho potomek. ListAdapter umí porovnávat změny datového zdroje a vláknem na pozadí.

ListAdapter potřebuje metody, jak porovnat dvě entity, zda jsou shodné. To realizujeme pomocí přetížím **DiffUtilItemCallback**, kdy doplníme porovnání, zda jsou dvě entity stejné. V tomto případě jsem zvolil porovnání pomocí atributu date.

ListAdapter potřebuje přepsat metody **onCreateViewHolder** a **onBindViewHolder**.

Ve funkci bind musíme dopsat, jak se atributy entity namapují na UI prvky.

```
class ScoreAdapter() : ListAdapter<Score, ScoreAdapter.ScoreViewHolder>(DiffCallback) {

    companion object {
        private val DiffCallback = object : DiffUtil.ItemCallback<Score>() {
            override fun areItemsTheSame(oldItem: Score, newItem: Score): Boolean {
                return oldItem.date == newItem.date
            }

            override fun areContentsTheSame(oldItem: Score, newItem: Score): Boolean {
                return oldItem == newItem
            }
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ScoreViewHolder {
        val viewHolder = ScoreViewHolder(
            ScoreRowBinding.inflate(
                LayoutInflater.from(parent.context),
                parent,
                false
            )
        )
        return viewHolder
    }

    override fun onBindViewHolder(holder: ScoreViewHolder, position: Int) {
        holder.bind(getItem(position))
    }

    class ScoreViewHolder(
        private var binding: ScoreRowBinding
    ): RecyclerView.ViewHolder(binding.root) {
        @SuppressLint("SimpleDateFormat")
        fun bind(score: Score) {
            binding.name.text = score.name
            binding.score.text = score.score.toString()
        }
    }
}
```

V aktivitě **ScoreActivity** musíme přidat inicializaci DAO.

```
private val scoreDao by lazy { ScoreDatabase.getDatabase(this).scoreDao() }
```

A následně provázat Recycler view s ListAdapterem.

```
binding.scoreRecyclerView.layoutManager = LinearLayoutManager(this)
val scoreAdapter = ScoreAdapter()
binding.scoreRecyclerView.adapter = scoreAdapter
lifecycle.coroutineScope.launch {
    scoreDao.getAllScores().collect() {
        scoreAdapter.submitList(it)
    }
}
```

Při stisknutí tlačítka by se měly zobrazit výsledky.

Git

Změny potvrdíme a uložíme do vzdáleného repository.

Naposledy změněno: čtvrtek, 2. května 2024, 15.30

◄ Tabulka nejlepších her (SQLite, Room) - video

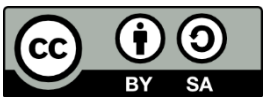
Přejít na...

Cvičení - úkol (čištění databáze, zobrazení výsledků) ►

Nápověda a dokumentace (anglicky)

UNIVERZITA PARDUBICE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Vytvořeno v rámci projektu **Digitalizace studijních Agend, Nové TechnologiE, systémy a přístupy k výuce na UPCE**, reg. č. NPO_UPCE_MSMT-16591/2022.



Financováno Evropskou unií
NextGenerationEU

Národní plán obnovy

MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY