Doporučené postupy v programování

Návrh procedur, funkcí a metod

Účel · Soudržnost · Délka a název · Práce s parametry · Vracení hodnot · Výjimky a chybové kódy

Lubomír Bulej

KDSS MFF UK

Základní terminologie

Rutina

- obecný koncept umožňující seskupovat související kód
- omezuje duplicitu, poskytuje základní nástroj pro abstrakci
- parametry, lokální proměnné, návrat do místa volání

Specifické druhy rutin

- Procedury
- Funkce
- Metody

Poznámky:

Primárním účelem funkcí je typicky spočítat nějakou hodnotu jako funkci vstupních parametrů. Opravdu čistých funkcí se v typickém programu asi najde málo, pokud to není program ve funkcionálním jazyce. Hlavní charakteristikou funkcí je absence vedlejších efektů, což má celou řadu výhod. Nicméně z pohledu běžného programátora se čistě funkcionální programování může jevit poměrně málo intuitivní.

Procedury jsou typické pro imperativní styl programování, kdy má programátor absolutní kontrolu nad průběhem výpočtu. Procedury obecně reprezentují aktivitu a výsledkem procedury není hodnota, ale např. přechod systému do jiného stavu. Přestože řada procedur vrací nějakou hodnotu a čistě syntakticky se jedná o funkce, svou povahou jsou to stále procedury.

Nakonec jsou metody, což jsou procedury a funkce svázané s daty.

Proč vytvářet procedury a funkce?

Omezení složitosti

- skrývání informací, abstrakce
  - flexibilita změny implementace
  - odsun zavlečené složitosti
- přiblížení řeči zadání – zlepšení čitelnosti
  - zjednodušení booleovských testů

Zamezení duplikace

- zlepšení udržovatelnosti
- možnost parametrizace
- obecnější koncept – **DRY**

Zvyšování výkonnosti

- důsledek abstrakce a zamezení duplikace

Poznámky:

V základních kurzech programování se člověk typicky dozví, že procedury a funkce slouží hlavně k zamezení duplikace kódu, cehož důsledek je především to, že se program lépe vyvíjí, ladí, dokumentuje a udržuje. Vedle syntaktických detailů jak používat parametry a lokální proměnné se toho člověk víc moc nedozví.

Přestože takové vysvětlení v každém bodě říká pravdu, zdaleka se nedotýká např. mnohem důležitějšího aspektu a tím je zvládání složitosti s využitím abstrakce a skrývání informací. Zamezení duplikace kódu samo o sobě typicky nepřináší všechny zmiňované výhody – ty právě plynou až z dobře strukturovaného programu, přimž vhodná struktura nevznikne sama od sebe pouhým naskládáním kódu do procedur a funkcí.

Často abstrahované operace

Sekvence

```
x = stack[topIndex];
topIndex--;

x = stackPop();
```

Práce s ukazateli

```
x = object_ptr->member;
object_ptr->member = y;

x = object_get_member(object_ptr);
object_set_member(object_ptr, x);
```

Přístup k atributům třídy

- výpočet on demand, logování, ...

Příklad: jméno posledního uzlu

Nedostatečná abstrakce

```
if (node != null) {
  Node currentNode = node;
  while (currentNode.next != null) {
    currentnode = currentNode.next;
  }

  leafName = currentNode.name;

} else {
  leafName = "";
}
```

Abstrakce skrývající mechanizmus

```
leafName = getLeafName (node);
```

Příklad: párovost operací zamykání

Součástí výkonného kódu

```
uintptr_t alloc_pages (...) {
  lock (pgalloc_lock);
  ...
  // do something
  ...
  if (status = FAILURE) {
    unlock (pgalloc_lock);
    return;
  }
  ...
  // do something else
  ...
  if (status = FAILURE) {
    return;
  }
  ...
  unlock (pgalloc_lock);
}
```

V nadřazené funkci

```
static
uintptr_t __alloc_pages (...) {
  ...
  // do something
  ...
  if (status = FAILURE) {
    return NULL;
  }
  ...
  // do something else
  ...
}

uintptr_t alloc_pages (...) {
  uintptr_t result;

  lock (pgalloc_lock);
  result =   alloc_pages (...);
  unlock (pgalloc_lock);

  return result;
}
```

Problém jednoduchých operací

V čem spočívá trivialita?

- složitost implementace neodráží složitost konceptu
  - inkrementace proměnné vs. přidělení unikátního identifikátoru

A jak je to s tím opakováním?

- pokud se kód neopakuje, neznamená to, že nepatří do funkce
- naopak, funkce dává prostor k pojmenování bloku kódu
  - vyšší úroveň abstrakce v místě volání, blok není nutné komentovat

A co s režií na volání funkce?

- počítejte s tím, že překladač i procesor je rozumný
- nesbírejte výkonnostní drobky, pokud to není nutné

Poznámky:

Pokud se opakují, není co řešit. Pokud se neopakují, dostáváme se do šedé zóny – je nutné zvážit zda se opakovat v principu mohou a jak ovlivňují soudržnost té konkrétní metody (viz. funkční soudržnost později).

Režie na volání funkce se není třeba bát. Pokud je funkce jednoduchá a překladač rozumný, předá parametry funkce v registrech, případně funkci přeloží inline. Volání funkce je něco jiného než podmíněný skok, takže i když je volání nepřímé, překladač může adresu skoku spočítat ve vhodný okamžik, takže v okamžiku zpracování instrukce skoku už je jasné, kam se bude skákat.

U moderních procesorů je navíc rozumné počítat s tím, že se snaží rozumně vykonávat operace spojené s voláním virtuálních metod, které jsou tak typické pro dnešní software.

Příklad: unikátní identifikátor

Nedostatečná abstrakce

```
public IdentifiedObject() {
  static int nextId = 0;
  this.id = nextId++;
  /* ... */
}
```

Abstrakce skrývající mechanizmus

```
private int getUniqueId() {
  static int nextId = 0;
  return nextId++;
}

public IdentifiedObject() {
  this.id = getUniqueId();
  /* ... */
}
```

Příklad: převod jednotek

Nedostatečná abstrakce

```
points = deviceUnits * (POINTS_PER_INCH / getDeviceUnitsPerInch ());
```

Abstrakce skrývající mechanizmus

```
int deviceUnitsToPoints (int deviceUnits) {
  return deviceUnits * (POINTS_PER_INCH / getDeviceUnitsPerInch ());
}

...
points = deviceUnitsToPoints (deviceUnits);
```

Při změně implementace

```
int deviceUnitsToPoints (int deviceUnits) {
  if (getDeviceUnitsPerInch () > 0) {
    return deviceUnits * (POINTS_PER_INCH / getDeviceUnitsPerInch ());
  } else {
    return 0;
```

```
    }
}
```

Jak nemá vypadat metoda?

Routine from Hell

```
void HandleStuff( CORP_DATA & inputRec, int crntQtr,
   EMP_DATA empRec, double & estimRevenue, double ytdRevenue,
   int screenX, int screenY, COLOR_TYPE & newColor,
   COLOR_TYPE & prevColor, StatusType & status, int expenseType )
{

int i;
for ( i = 0; i < 100; i++ ) {
   inputRec.revenue[i] = 0;
   inputRec.expense[i] = corpExpense[ crntQtr ][ i ];
   }
UpdateCorpDatabase( empRec );
estimRevenue = ytdRevenue * 4.0 / (double) crntQtr;
newColor = prevColor;
status = SUCCESS;

if ( expenseType == 1 ) {
     for ( i = 0; i < 12; i++ )
           profit[i] = revenue[i] - expense.type1[i];
     }
else if ( expenseType == 2 ) {
          profit[i] = revenue[i] - expense.type2[i];
          }

else if ( expenseType == 3 )
          profit[i] = revenue[i] - expense.type3[i];
          }
```

Jak má tedy vypadat správná metoda?

Správná metoda vykazuje silnou soudržnost

- dělá právě jednu věc, dělá ji dobře a nedělá nic jiného
- minimální závislost na okolí a pořadí volání

Soudržnost/koheze (cohesion)

- vzájemná souvislost operací v kódu metody
- `Cosine()` vs. `CosineAndTan()`
- u návrhu tříd soudržnost nahrazena abstrakcí a zapouzdřením

Silně soudržné metody vykazují méně chyb

- Card, Church, Agresti (1986) – ze 450 rutin bylo bez chyby 50% resp. 18% silně resp. slabě soudržných funkcí
- Selby, Basili (1991) – nejslaběji soudržné rutiny 7× chybovější a 20× nákladnější na opravu

Jak se pozná silná a slabá soudržnost?

Ideální forma soudržnosti

- funkční soudržnost
  - metoda dělá právě jednu věc odvoditelnou z názvu

- `sin()`, `getCustomerName()`, `eraseFile()`, `ageFromBirthday()`, ...

Méně ideální formy soudržnosti

- sekvenční soudržnost
  - operace musí být v určitém pořadí, v jednotlivých krocích sdílí mezivýsledky
  - příklad: výpočet věku z data narození následovaný výpočtem doby do důchodu z věku
- komunikační soudržnost
  - operace používají stejná data, ale jinak spolu nesouvisí
  - příklad: rutina vytiskne a reinicializuje sumární data
- časová soudržnost
  - operace spolu nesouvisí, jen mají být vykonány ve stejné fázi běhu
  - příklad: posloupnost různých operací při inicializaci programu
  - problém najít název → koordinační metoda – pouze volá jiné metody

Jak se pozná silná a slabá soudržnost?

Nevyhovující formy soudržnosti

- procesní soudržnost
  - operace spolu nesouvisí, jejich pořadí určeno nesouvisejícím procesem
  - `getEmployeeData()` vs `getFirstPartOfEmployeeData()`, `getRestOfEmployeeData()`
- logická soudržnost
  - výběr z více nesouvisejících operací pomocí řídícího parametru
  - přiléhavější termín by byl *nelogická soudržnost*
  - výjimkou jsou speciální metody → *event handler*, *request dispatcher*
- nahodilá (žádná) soudržnost
  - operace spolu nijak nesouvisí → nulová/chaotická soudržnost
  - příklad: routine from hell

Závislost na okolí a pořadí volání

Minimalizace závislosti na okolí

- co nejsamostatnější kus kódu
- obecně práce pouze daty předanými volajícím
  - u metod implicitně předáván parametr `this`

Minimalizace závislosti na pořadí volání

- speciální případ minimalizace stavového prostoru
- pokud se závislosti nelze vyhnout
  - vytáhnout ji na světlo – učinit ji explicitní
  - přidat vysvětlující komentář – jak a proč
  - obrnit se proti špatnému použití – defenzivní programování
- typicky problematické oblasti: multithreading, synchronizace

Příklad: explicitní závislost

Bez zjevné závislosti

```
computeMarketingExpense (marketingData)
computeSalesExpense (salesData)
computeTravelExpense (travelData)
computePersonnelExpense (personnelData)
displayExpenseSummary (
    marketingData, salesData, travelData, personnelData)
```

Se zjevnou závislostí

```
expenseData = initializeExpenseData (expenseData)
expenseData = computeMarketingExpense (expenseData)
expenseData = computeSalesExpense (expenseData)
expenseData = computeTravelExpense (expenseData)
expenseData = computePersonnelExpense (expenseData)
displayExpenseSummary (expenseData)
```

Příklad: komentovaná explicitní závislost

Se zjevnou závislostí a komentářem

```
//
// Compute expense data. Each of the routines accesses the member
// data expenseData. DisplayExpenseSummary should be called last
// because it depends on data calculated by other routines.
//
expenseData = initializeExpenseData (expenseData)
expenseData = computeMarketingExpense (expenseData)
expenseData = computeSalesExpense (expenseData)
expenseData = computeTravelExpense (expenseData)
expenseData = computePersonnelExpense (expenseData)
displayExpenseSummary (expenseData)
```

Poznámky:

Komentář k závislosti upozorňuje na nutnost specifického uspořádání a na riziko špatného výsledku při jeho nedodržení, ale není z něj jasné, proč je závislost (obzvláště v tomto případě) vůbec nutná. Pokud takový návrh nejsme schopni zdůvodnit, je pravděpodobně lepší se mu vyhnout.

Příklad: komentovaná závislost

Bez zjevné závislosti s komentářem

```
//
// The following calls must be in correct order. Inside the
// taskReachedState method, the Task Manager may decide to
// close the context which contains this task by calling the
// HostRuntimeInterface.closeContext method.
//
// If the calls are not properly ordered, the Host Runtime will not
// have been notified about the task's completion (the notification
// happens in the notifyTaskFinished call) and will refuse to close
// the context (throwing IllegalArgumentException).
//
// This would lead to a race condition.
//
hostRuntime.notifyTaskFinished(TaskImplementation.this);
hostRuntime.getHostRuntimesPort().taskReachedState(
  taskDescriptor.getTaskTid(),
  taskDescriptor.getContextId(),
  processKilledFromOutside
    ? TaskState.ABORTED
```

```
        : TaskState.FINISHED
    );
```

Poznámky:
Příklad je převzat ze softwarového projektu Davida Majdy. Pro demonstrační účely není podstatné, co přesně dělají metody `notifyTaskFinished` a `taskReachedState`, ale to, jak podrobně je chování v komentáři popsáno.

Délka procedur a funkcí

Studie chybovosti v závislosti na délce metody

- délka negativně koreluje s chybovostí; s rostoucí délkou (do 200 řádek) klesá chybovost (počet chyb na řádek kódu)
- chybovost nesouvisí s délkou, ale se strukturální složitostí a množstvím dat, se kterými kód manipuluje
- u krátkých rutin (méně než 32 řádek) není korelace mezi délkou a nižší chybovostí
- studie 450 rutin ukázala, že menší rutiny (s méně než 143 příkazy) vykazovaly větší počet chyb na řádek než delší rutiny, ale jejich oprava byla 2.4× méně nákladná
- jiná studie ukázala, že kód bylo nutné měnit nejméně, když byly rutiny dlouhé 100-150 řádků
- studie IBM ukázala, že rutiny nejvíce náchylné k chybám byly delší než 500 řádků; za touho hranicí chybovost rostla úměrně s délkou

Délka procedur a funkcí

Rozumná délka

- méně než 100–200 řádků
  - délka nemá velký vliv na chybovost
  - přesto má vliv na pohodlí práce programátora
  - levnější vývoj (nižší cena za řádek)
  - i při mírně vyšší chybovosti levnější opravy
- soft limit – 2 obrazovky textu (50–60 řádků)
  - v případě potřeby nechat růst k hranici 200 řádků
  - důležitá je strukturální složitost, počet proměnných, hloubka zanoření
    - počet rozhodovacích bodů – řídící struktury a logické spojky

Názvy procedur a funkcí

Název by měl přesně a úplně popisovat, co procedura dělá nebo co funkce vrací.

Obecné požadavky

- orientace na problém

- délka názvu podle potřeby
  - procedury a funkce složitejší než proměnné → delší názvy

Souvislost názvu s návrhem procedur a funkcí

- problém vymyslet název ~ nejasný úkol
- název obsahuje spojky (ano, or....) ~ dělá více věcí
- příliš dlouhý název ~ dělá moc věcí

Tvorba názvu procedur a funkcí

Typický formát

- *přísudek* – [*přívlastky*] – *předmět*, rozkazovací způsob
  - `getID`, `computeEquationResults`, `drawWindowBorder`
- u funkcí předmět pokud možno návratová hodnota
- pozor na nic neříkající slovesa: handle, perform, process, do,...

Upřednostňujte krátká jména pokud to jde

- příklad: starý a nový iterátor v Javě
  - `Enumerable.hasMoreElements` vs `Iterator.hasNext`
  - `Enumerable.nextElement` vs `Iterator.next`
- třída = kontext → umožňuje vynechat některé přívlastky
  - `User.getUserName` vs `User.getName`
  - vyžaduje odpovídající název proměnné/parametru

Tvorba názvu procedur a funkcí

Použití správných antonym

- add/remove
- begin/end
- create/destroy
- first/last
- get/put
- get/set
- increment/decrement
- insert/delete
- lock/unlock
- min/max
- next/previous (prev – symetrické)
- old/new
- open/close
- show/hide
- source/target
- start/stop
- up/down

Parametry procedur a funkcí

Význam parametrů

- parametrizace – oddělení kódu a dat

- seznam parametrů určuje rozhraní funkce

Druhy parametrů

- role parametru
  - vstupní
  - modifikovatelné – vstupně/výstupní
  - výstupní
- způsob předání
  - hodnotou
  - odkazem

Obvyklé korelace

- vstupní ~ hodnotou
- modifikovatelné, výstupní ~ odkazem

Zásady pro práci s parametry

Vyhýbejte se modifikovatelným a výstupním parametrům

- v místě použití (a většinou i v deklaraci) se težko odlišují od vstupních
  - Java – primitivní typy vždy hodnotou, objekty vždy odkazem
  - C++ – primitivní typy hodnotou nebo ukazatelem, objekty hodnotou, ukazatelem, referencí

Snažte se použít mechanizmus návratové hodnoty

- bližší představě matematické funkce
- některé jazyky (Python, Ruby, PHP, …) umožňují vrátit více hodnot

```
a, b = divmod(7, 3)
print a  # => 2
print b  # => 1
```

- v ostatních jazycích (C/C++, Java, C#, …) vrátit strukturu/objekt
  - speciálně při vracení více než 2 hodnot

Zásady pro práci s parametry

Situace vhodné k použití modifikovatelných a výstupních parametrů

- obecně při nutnosti vracet doplňkové informace
  - výsledek operace a výsledná hodnota – nepřetěžuje význam vracených hodnot
  - Java – nutnost *holder* objektů
- specificky např. pokud je za alokaci paměti pro data zodpovědný volající

Použití modifikovatelných a výstupních parametrů

- v případě nutnosti se snažte omezit na 1 takový parametr
  - bud čistě návratová hodnota, nebo dodatečná stavová informace
- dodržujte pořadí vstupní – modifikované – výstupní

- snažte se odlišit vstupní parametry od modifikovaných a výstupních
- použití parametrů řádně zdokumentujte

Zásady pro práci s parametry

Podobné parametry uvádějte ve stejném pořadí u všech funkcí

```
fprintf(stream, format,...)
fputs(str, stream)

strncpy(dst, src, len)
memcpy(dst, src, len)
```

Vyhněte se nepoužívaným parametrům

- stejné jako u nepoužívaných proměnných
- výjimkou je nutnost dodržení signatury
  - callbacky, zpětná kompatibilita, apod.

Neměňte vstupní parametry v těle metody

- sémantika parametru je zachována, umožňuje extrahovat blok kódu do metody
- Java – `final`, C/C++ – `const`

Poznámky:

Klíčová slova `const` resp. `final` často znepřehledňují kód, a tak je málokdo používá. Při návrhu nových jazyků by možná stálo za úvahu dát parametrům sémantiku konstant automaticky, bez potřeby klíčového slova.

Příklad: použití vstupních parametrů

Vstupní parametr supluje pomocnou proměnnou

```
float response (float inputSample) {
  sampleHistory.store (inputSample);

  inputSample = 0.0f;
  for (int i = 0; i < coefficients.length; i++) {
    ...
    inputSample += coefficients [i] * sampleHistory.previous (i);
    ...
  }
  ...
  return inputSample;
}
```

Příklad: použití vstupních parametrů

Vstupní parametry jsou tabu (pro modifikaci)

```
float response (final float inputSample) {
  sampleHistory.store (inputSample);

  float outputValue = 0.0f;
  for (int i = 0; i < coefficients.length; i++) {
    ...
```

```
        outputValue += coefficients [i] * sampleHistory.previous (i);
        ...
    }
    ...
    return outputValue;
}
```

Zásady pro práci s parametry

Preferujte abstraktní typy

- rozhraní (interface) v jazyce Java
  - `void printNames (ArrayList <String> names);`
  - `void printNames (List <String> names);`
- platí i pro návratové hodnoty

Vyhněte se booleovským parametrům

- v místě volání není vidět, co `true` nebo `false` znamená
  - `int compareStrings (String s1, String s2, boolean caseInsensitive)`
- výjimkou jsou metody, kde je význam zřejmý z názvu
  - `setEnabled (boolean enabled)`
- nahradit výčtovým typem/konstantami nebo dvojicí metod

Poznámky:

K interfacům a abstraktním typům: Viz *Effective Java: Programming Language Guide, Item 34*.

K booleovským parametrům: Náhrada výčtovým typem/konstantou má i výhodu flexibility v případě, že časem přibudou další možné hodnoty parametru. To se u boolovských parametrů docela často stává, více viz výstižně nazvaný článek Booleans suck.

Zásady pro práci s parametry

Vyhněte se použití více než 5–7 parametrů

- v případě potřeby použijte strukturovaný typ s vhodnou úrovní abstrakce
- velké množství předávaných parametrů indikuje příliš silnou vazbu

Předávejte parametry odpovídající poskytované abstrakci

- klasický problém – předat specifické hodnoty vs. celý objekt
  - předání celého objektu porušuje zapouzdření a zesiluje vazbu
  - předání celého objektu zvyšuje stabilitu rozhraní
- důležitá je abstrakce, kterou funkce poskytuje
  - pokud očekává hodnoty, které jsou náhodou také v objektu, který ovšem s funkcí nesouvisí, předávejte hodnoty
  - pokud metoda očekává, že dostane do ruky objekt a něco s ním udělá, předávejte objekt

Poznámky:

Při zkracování seznamu parametrů funkce náhradou několika atributů jednoho objektu celým objektem je dobré se zamyslet, zda je fakt, že funkci chcete posílat celý objekt náhoda nebo systematická záležitost. Náhoda se pozná tak, že při volání funkce nemáte vždy dotyčný objekt "v ruce" – v tom případě je lepší nechat seznam parametrů být, protože budete muset u některých volání funkce objekt uměle vytvářet. Naopak, pokud zjistíte, že to náhoda není, stojí za to se chvilku zastavit nad strukturou kódu – můžete např. přijít na to, že daná funkce by měla být metoda objektu, který jí posíláte.

Návratové hodnoty

Procedura vs. funkce

- vhodné rozlišovat syntaktický a logický pohled
- funkce vrací hodnotu odvozenou ze vstupních parametrů
  - někdy také z kontextových parametrů uvnitř objektu
- procedura často vrací hodnotu reprezentující výsledek operace
  - hodnota ovšem často reprezentuje stavovou informaci

Návratové hodnoty

Nepřetěžujte význam návratové hodnoty funkcí

- název funkce by měl popisovat vracenou hodnotu
- výsledek funkce by nemělo být nutné testovat na úspěch
  - funkce by neměla mít vedlejší efekty
- pro neplatné hodnoty použijte `NaN` nebo výjimky
  - `atoi()`, `atol()`, `atoll()`

Vyhněte se používání procedur ve výrazech

- výraz by měl být jako funkce – bez vedlejších efektů
- zabudování do výrazu znesnadňuje modifikaci kódu
- návratové hodnoty jsou často významově přetížené
  - stavová informace přímo nesouvisí s primární funkcí

Příklad: testování výsledku procedury

Nevhodné použití procedury ve výrazu

```
if (report.formatOutput (formattedReport) == FormatResult.SUCCESS) {
  ...
}
```

Zdůraznění procedurálního charakteru metody

```
formatResult = report.formatOutput (formattedReport);
if (formatResult == FormatResult.SUCCESS) {
  ...
}
```

Znemožnění použití procedury ve výrazu (v Javě nešikovné)

```
report.formatOutput (formattedReport, resultHolder);
if (resultHolder.result == FormatResult.SUCCESS) {
  ...
}
```

Obecná doporučení pro vracení hodnot

Předčasný návrat používejte s rozvahou

- obecně komplikuje control-flow – porušuje princip single-entry/single-exit
- výjimky: strážné podmínky na začátku, konce nezávislých větví

Používejte jednotný název pro proměnnou s výsledkem

- prefix vhodnější, např. `result`
  - analogie se značením input/output parametrů

Vracejte prázdné kontejnery místo `null`

- pole, seznamy a kolekce obecně
- obsah se prochází, kód funguje i s prázdným kontejnerem
  - nefunguje s `null` kontejnerem: `HttpServletRequest.getCookies()`
- test na prázdnost má mírně jiný význam než test na `null`

Poznámky:
K prázdným polím vs. `null` viz *Effective Java: Programming Language Guide, Item 34*.

S podporou pro funkcionální programování přibyla v Javě od verze 8 třída `Optional`, což je (immutable) kontejner na 1 hodnotu.

Předávání doplňkových informací

Hlavní typy doplňkových informací

- úspěch/selhání operace
- příčina selhání operace

Hlavní způsoby předávání doplňkových informací

- platnost/neplatnost návratové hodnoty
  - pro kódování příčiny neúspěchu vyžaduje množinu neplatných hodnot
  - prakticky použitelné pro celočíselné typy, náročné na údržbu
- výstupní parametr obsahující stavovou informaci

    - Java – složité použití, režie malých objektů
- strukturovaná návratová hodnota/výstupní parametr
    - sdružuje návratovou hodnotu a stavovou informaci
    - málo obvyklé, režie malých objektů
- výjimka nesoucí informace o selhání i příčině

Příklad: výstupní stavová proměnná

Obecná třída pro výstupní stavové proměnné

```
public final class StatusHolder <S> {
  public S status;
}
```

Použití s konkrétním typem stavové informace

```
QueryResult DB.execute (
  Query query, StatusHolder <QueryStatus> statusHolder);
...
StatusHolder <QueryStatus> queryStatusHolder =
  new StatusHolder <QueryStatus> ();
QueryResult queryResult =
  db.execute (findInactiveUsersQuery, queryStatusHolder);

if (queryStatusHolder.status == QueryStatus.SUCCESS) {
  ...
  for (Row row : queryResult.rows ()) {
    ...
  }
}
```

Příklad: strukturované návratová hodnota

Obecná třída pro (objektové) návratové hodnoty

```
public final class StatusResult <S, R> {
  public final S status;
  public final R result;

  public StatusResult (S status, R result) {
    this.status = status;
    this.result = result;
  }
}
```

Použití s konkrétním návratovým a stavovým typem

```
StatusResult <QueryStatus, QueryResult> DB.execute (Query query);
...
StatusResult <QueryStatus, QueryResult>
  queryStatusResult = db.execute (findInactiveUsersQuery);

if (queryStatusResult.status == QueryStatus.SUCCESS) {
  ...
  for (Row row : queryStatusResult.result.rows ()) {
    ...
  }
}
```

Příklad: výjimka

Třída vyjímek pro zpracování dotazu

```
public class QueryException extends ... {
  ...
```

```
}
```

Odchycení výjimky při selhání

```
QueryResult DB.execute (Query query) throws QueryException;
...
try {
  QueryResult queryResult = db.execute (findInactiveUsersQuery);
  for (Row row : queryResult.rows ()) {
    ...
  }
  ...
} catch (QueryException e) {
  ...
}
```

Výjimky vs. chybové kódy

Výjimky

- přirozené médium pro doplňkové informace o chybě
  - umožňuje předat informace o selhání i jeho příčině
  - nevyžaduje přetěžování významu návratových hodnot
  - nevyžaduje výstupní parametry a strukturované návratové hodnoty
- pokud se neodchytí, propagují se do vyšších vrstev
  - usnadňují návrat z hluboce vnořených volání
- kompilátor může vynutit odchycení

… ale …

- v místě volání funkce nejsou vidět
- většinou mají netriviální běhovou režii
- na jejich používání není jednotný názor

Výjimky vs. chybové kódy

Chybové kódy

- typicky nutí k přetěžování významu návratových hodnot
- v místě volání nemusí být vidět
  - pokud nejsou předávány ve výstupním parametru nebo strukturované návratové hodnotě, což zase typicky zvyšuje míru zavlečené složitosti
- musí se ošetřit na místě, ale nic k ošetření nenutí
- musí se explicitně propagovat do vyšších vrstev

… ale …

- jsou používány už dlouhá léta
- dají se použít vždy
- nemají výraznější režii

Poznámky:

Problematika výjimky vs. chybové kódy je složitější – my jsme se jí tu jen zlehka dotkli. Zájemcům o motivační diskuzi lze doporučit k přečtení následující články (v uvedeném pořadí):

1. Joel Spolsky: Exceptions
2. Ned Batchelder: Exceptions vs. Status Returns
3. Joel Spolsky: DoSomething()
4. Ned Batchelder: Exceptions in the Rainforest

Ned Batchelder má hezký model toho, jak vypadá software v článku "Exceptions in the rainforest". Podle něj má software 3 vrstvy, shora C-B-A. Nejnižší (A = Adapting software beneath) přizpůsobuje jiný kód našim potřebám. Někdy jsou to low-level volání, pak se často používají chybové kódy, které je dobré převádět na výjimky.

Prostřední vrstva (B = Building pieces of your system, někdy také Business logic :-) slouží k vytvoření částí, ze kterých se skládá náš svět. Tady je prostor pro problem-specific koncepty, algoritmy a datové struktury. V prostřední vrstvě chceme být maximálně produktivní a chceme tady mít dobře čitelný kód.

Nejvyšší vrstva (C = Combining it all together) ví co se děje, takže typicky ví, co dělat s výjimkami. Výjimky neznamenají, že error handling bude najednou snadnější, ale znamenají, že chyby z vrstvy A se neztratí, a že nemusíme "špinit" vrstvu B tím, že bude předávat chyby výše, do vrstvy C.

Typicky pak vrstva A vesměs vyhazuje výjimky, vrstva B také, ale méně, a vrstva C primarne chytá výjimky a potenciálně něco užitečného dělá.

A protože software je fraktální, dá se tenhle model embeddovat do různých vrstvev.

Výjimky vs. chybové kódy

Kdy je vhodne používat výjimky?

- jasným kandidátem jsou opravdové funkce
  - funkce se používají ve výrazech, výsledky se explicitně netestují
  - k selhání může dojít v podstatě pouze v důsledku chyby programátora
  - většinou na ně nelze rozumně reagovat, podobné signálům
- procedury pracující s okolím
  - stav okolního prostředí se může měnit asynchronně
  - k selháním nedochází v důsledku programových chyb
  - reakce je součástí návrhu → hlavní předmět "sporu" o výjimky
  - při prototypování je snadné obsluhu odložit, problém neztratí
- obecně: pokud to zjednoduší nebo zpřehlední kód/návrh
  - nehodí se přetěžování významu návratové hodnoty
  - jiné mechanizmy předávání doplňkových informací jsou neohrabané

Poznámky:

Co se týče toho zda výjimky používat, pro nastartování diskuze postačí již dříve zmiňovaná debata Joel Spolsky vs. Ned Batchelder. K tomu je např. zajímavá polemika na obhajobu chybových kódů od Douga Rosse:

- Return-codes vs. Exceptions, Part 129
- Return-codes vs. Exceptions, Part 318
- Exceptions, Tunable Logging, and ...
- Return-codes vs. Exceptions, Part 516

Základní pravidlo asi jako obvykle zní – pokud použití výjimek zjednoduší a zpřehlední kód, asi není co řešit. Důležité je pak používat výjimky správně používat. Nakonec asi není až tak podstatné, zda se používá to či ono, ale zda se programátor vědomě a systematicky věnuje obsluze abnormálních stavů.

Zásady pro práci s výjimkami

Výjimky používejte jen ve výjimečných situacích

- výjimečné v daném kontextu – nepoužívat výjimky pro běžný control flow

Co jsou určitě chyby?

- nesplněné preconditions
- nesplněné postconditions – např. platná návratová hodnota
- porušení invariantů třídy

Poznámky:

K výjimečným situacím: Viz *Effective Java: Programming Language Guide, Item 39: Use exceptions only for exceptional conditions.* To znamená nepoužívat výjimky pro řízení toku programu, jako např. vynechání kontroly `Iterator.hasNext()` a čekání na to až `Iterator.next()` vyhodí výjimku.

Příklad: použití výjimek pro řízení toku

Cyklus ukončen výjimkou

```
...
try {
  while (true) {
    Employee employee = employeeIterator.next ();
    ...
  }
catch (NoSuchElementException e) {
}
```

Správné použití iterátoru

```
...
while (employeeIterator.hasNext ()) {
  Employee employee = employeeIterator.next ();
  ...
}
```

Jak poznat výjimečné případy?

Specifikujte kontrakt metody

- požadavky na parametry a kontext (preconditions)
  - je parametr vstupní/modifikovatelný/výstupní?
  - smí být `null`?
  - rozsahy číselných parametrů (speciálně kladnost/nezápornost, není-li k dispozici `unsigned`)
  - jednotky
- vlastnosti návratové hodnoty (postconditions)
- vedlejší efekty a možnosti selhání
- předpokládaný způsob použití

Kam umístit specifikaci kontraktu?

- do dokumentace
- do kódu – defenzivní programování

Poznámky:
Viz *Effective Java: Programming Language Guide, Item 23*.

Zásady pro práci s výjimkami

Používejte výjimky na odpovídající úrovni abstrakce

- systematické
- flexibilita změny implementace
- řetězení výjimek – exception chaining

Ve výjimce detailně popište problém

- Včetně případných chybných hodnot parametrů/proměnných

Vyhněte se prázdným `catch` blokům

- pokud "by nemělo nikdy nastat", vložit kód vyvolávající chybu
- typicky assertion, více defenzivní programování (později)

Poznámky:

K úrovni abstrakce: Viz *Effective Java: Programming Language Guide, Item 43: Throw exceptions appropriate to the abstraction*. To znamená, že výjimky, které metoda vyvolává by měly být na

úrovni abstrakce odpovídající tomu, co metoda dělé, ne jak to dělá.

K detailnímu popisu problému: Viz *Effective Java: Programming Language Guide, Item 45*.

K prázdným `catch` blokům: Viz *Effective Java: Programming Language Guide, Item 47*.

Příklad: úroveň abstrakce výjimek

Metoda deklaruje výjimku závislou na implementaci

```
class Employee {
  ..
  public TexId getTaxId () throws IOException {
    ...
  }
  ...
}
```

Zřetězení výjimek pro změnu úrovně abstrakce

```
class Employee {
  ...
  public TexId getTaxId () throws EmployeeDataNotAvailableException {
    ...
    try {
      ...
    catch (IOException e) {
      throw new EmployeeDataNotAvailableException (e);
    }
  }
  ...
}
```

Java a výjimky

Jazyková podpora

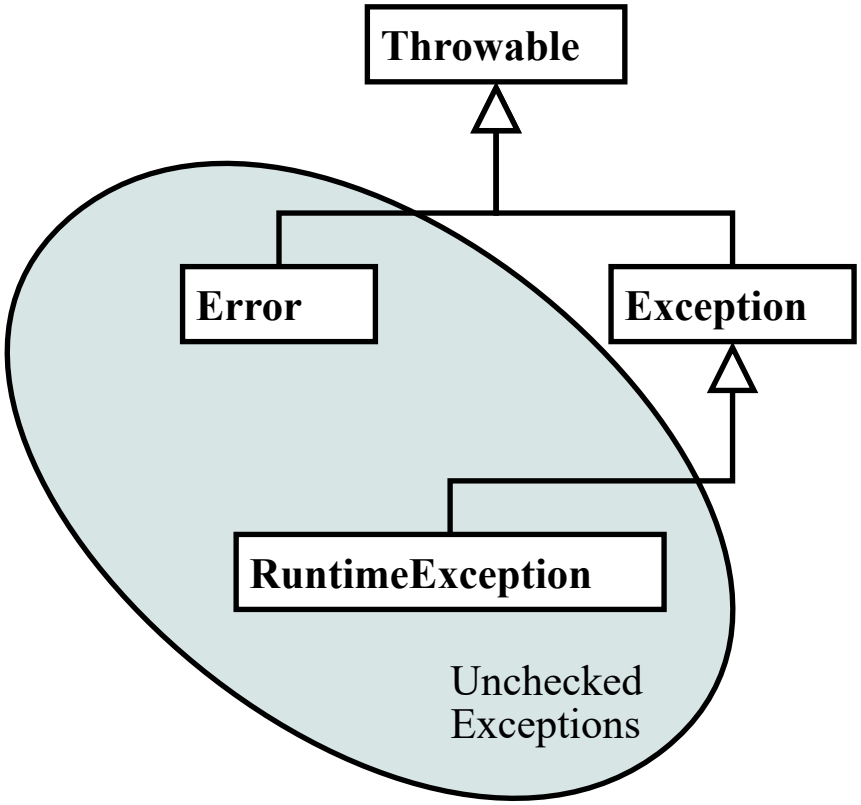
- příkaz `throw`, slouží k vyvolání výjimky
- konstrukce `try-catch`, slouží k odchycení třídy výjimek
- konstrukce `try-finally`, slouží k uvolnění prostředků při výjimce

Hierarchie výjimek

Druhy výjimek

- checked
  - musí být v signatuře metody
  - volající je musí povinně odchytit, nebo deklarovat v signatuře
  - kontrolováno staticky při překladu

- unchecked



Kontrolované vs. nekontrolované výjimky

Snaha rozlišit dva druhy výjimek

- výjimky jako součást rozhraní metody
  - očekává se, že na ně bude volající reagovat
  - reakce se vynucuje statickou kontrolou překladačem
- výjimky jako důsledek běhových chyb
  - volající na ně může a nemusí reagovat
  - reakce se nevynucuje

Důsledky nadužívání/nevhodného používání kontrolovaných výjimek

- polykání výjimek – prázdné `catch` bloky
  - snaha o vynucení obsluhy vede k maskování chyb
- neošetřené vyjímky – propagace do signatur metod
  - změna v deklarovaných kontrolovaných výjimkách znemožní překlad klientského kódu

Kontrolované vs. nekontrolované výjimky

Kdy používat kontrolované a nekontrolované výjimky?

- kontrolovaná výjimka vynucuje obsluhu u volajícícho
  - pokud může nastat i při správném použití metody a dá se očekávat, ze volající ji může rozumně ošetřit
  - pokud si nejste jisti nebo záleží na kontextu použití → nekontrolované
- nekontrolované – chyby programátora

- kontrola parametrů, indexů, ...
- fatální chyby – něco je strašně špatně
- výjimky, které bude obsluhovat jen málo volajících
- zvýšený důraz na dokumentaci

Poznámky:

Mnohem těžší na rozhodnutí je dilema kolem kontrolovaných a nekontrolovaných výjimek. Řada expertů je dnes považuje za omyl a doporučují vyhazovat pouze nekontrolované výjimky a kontrolované převádět na nekontrolované. Viz např. Bruce Eckel:

- Does Java Need Checked Exceptions?

Celou debatu pak ale poměrně dobře shrnuje Brian Goetz na webu IBM developerWorks:

- Java theory and practice: The exceptions debate

A poměrně dobrý návod poskytuje také Barry Ruzek na serveru Dev2Dev (odkaz vede jinam, puvodní server je nedostupný):

- Effective Java Exceptions

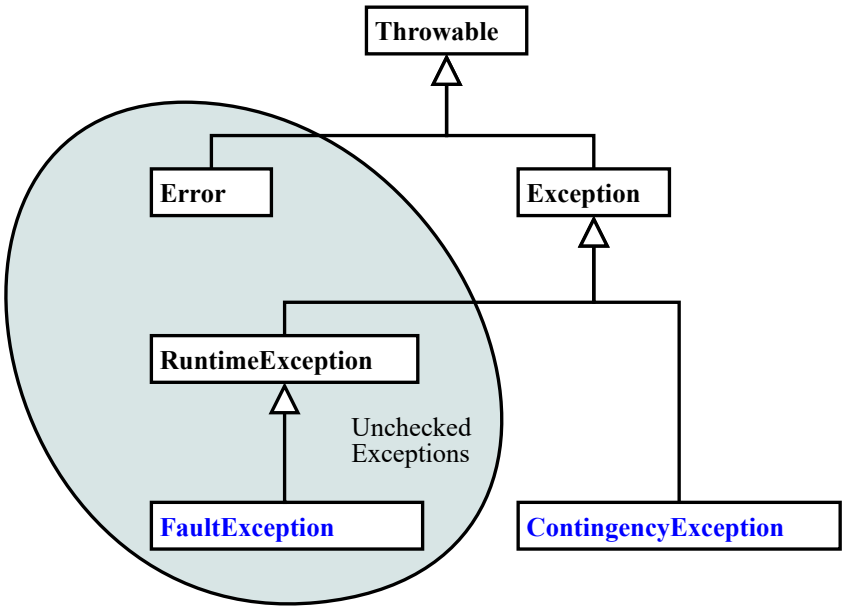
Mapování výjimek do Javy

Eventuality a chyby

eventualita (contingency)
: Výjimečná, ale předpokládaná situace, která se dá popsat v pojmech odpovídajících zamýšlenému účelu funkce, a která vyžaduje alternativní zpracování na úrovni volajícího. Volající je na takovou situaci připraven a má strategii pro jejich řešení.

chyba/porucha (fault)
: Neplánovaná situace, která metodě znemožnila vykonat zamýšlenou funkci, a která nemůže být popsána bez znalosti implementace funkce.

Hierarchie výjimek



Jaké výjimky použít?

Situace	Eventualita/Contingecy	Chyba/Fault
Považováno za	součást návrhu	ošklivé překvapení
Očekávaný výskyt	předvídatelný, ale vzácný	nikdy
Koho to zajímá	kód volající metodu	lidi, kteří mají odstranit problém
Příklad	alternativní návratové hodnoty	programové chyby, selhání hardware, konfigurační chyby, chybějící soubory, nedostupné servery
Nejlepší mapování	kontrolovaná výjimka	nekontrolovaná výjimka

Poznámky:

Zdroj: Barry Ruzek: Effective Java Exceptions

Kontrolované vs. nekontrolované výjimky

Kontrolované výjimky vyvolávejte v situacích, kdy je možné zotavení

- vyhněte se zbytečnému používání kontrolovaných výjimek v situacích, ze kterých se volající nemůže dost dobře zotavit
- častá transformace: kontrolovaná výjimka → testovací metoda + nekontrolovaná výjimka

Nekontrolované výjimky vyvolávejte při programových chybách

- nesplněné preconditions metody, fatální chyby, ...

Poznámky:

K náhradě kontrolovaných výjimek za nekontrolované: Dobrý příklad je metoda parsující text v nějakém jazyce. Text může v principu obsahovat chyby, a je žádoucí, aby metoda v tom případě vyhazovala výjimku. Pokud by tato výjimka byla kontrolovaná, musel by programátor výjimku odchytávat i v případě, že by si byl jist, že metodě předává syntakticky korektní text (např. automaticky generovaný). Lepší řešení je použít nekontrolovanou výjimku a přidat metodu, která bezchybnost textu otestuje a vrátí `true` nebo `false`.

Uvedený postup ale nelze použít vždy. Např. pokud bychom chtěli před smazáním souboru zkontrolovat, zda tento soubor existuje, může nám ho mezi testem a smazáním někdo "smazat pod rukama". Je tedy potřeba mít zaručen exkluzivní přístup k datům.

Ke kontrolovaným a nekontrolovaným výjimkám: Viz *Effective Java: Programming Language Guide, Item 40: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors*.

K užívání kontrolovaných výjimek: Viz *Effective Java: Programming Language Guide, Item 41: Avoid unnecessary use of checked exceptions*.

Osobně se přikláním spíše k názoru, že kontrolované výjimky mohou být užitečné primárně ve vlastním kódu, kde si tím mohu zajistit, že nezapomenu na obsluhu nějakého chybového stavu. Vně svého kódu bych kvůli minimalizaci obtěžování konzumentů mého kódu vyhazoval výjimky nekontrolované. Ty je pak nutné velmi dobře zdokumentovat. Pro situace, kdy je možné provést test před voláním výkonné metody (a kdy se situace po volání testu nemůže změnit), bych se snažil mít k výkonným metodám vždy testovací metody. Výkonné metody by pak mohly vyhazovat nekontrolované výjimky (bylo by chybou programátora, že nezkontroluje, zda může výkonnou metodu volat). Pokud není možné test a vykonání operace rozumně oddělit, volil bych spíše nekontrolovanou výjimku, abych nezatěžoval konzumenta. Dá se očekávat, že konzument nejspíš bude můj kód nějak adaptovat a kontrolované výjimky si může na vlastním území zavést sám.

Kontrolované vs. nekontrolované výjimky

Příklad: `java.io.RandomAccessFile`

```
public int read (byte [] b, int off, int len)
  throws IOException
```

Příklad: `java.io.DataInputStream`

```
public final void readFully (byte[] b, int off, int len)
  throws EOFException, IOException

public final double readDouble ()
  throws EOFException, IOException
```

- `IOException` nebo `EOFException`

Kontrolované vs. nekontrolované výjimky

Příklad: `java.util.Arrays`

```
public static <T>
int binarySearch (T [] a, T key, Comparator <? super T> c)
```

Příklad: `java.rmi.Naming`

```
public static Remote lookup (String name)
  throws NotBoundException, MalformedURLException, RemoteException
```

Poznámky:

Zde stojí za zmínku, že obvykle pokud něco hledáme, tak počítáme s možností, že to nenajdeme. Proč je to v případě RMI lookupu jiné? Stejně tak pokud víme, že zadané URL je správně, tak proč muset řešit checked exception?

Další doporučení pro práci s výjimkami

Nevyhazujte výjimky z konstruktorů/destruktorů

- v C++ se nevolá destruktor, pokud konstruktor nedoběhl
- příležitost pro únik prostředků (resource leaks)
- obecně: nedělejte v konstruktorech práci

Snažte se využívat standardní výjimky jazyka

- Java – výjimky z `java.lang.*`, `java.util.*`, …
  - `IllegalArgumentException`, `IllegalStateException`
  - `NullPointerException`
  - `IndexOutOfBoundsException`
  - `ConcurrentModificationException`
  - `UnsupportedOperationException`
  - …
- musí odpovídat poskytované abstrakci a kontextu

Poznámky:

K standardním výjimkám: Viz *Effective Java: Programming Language Guide, Item 42*.