Grundlagen der Programmkonstruktion

Kontrollfragen - 4. Test

Kapitel 6

Welche beiden grundlegenden Speicherbereiche werden in Java (und fast allen anderen Programmiersprachen) unterschieden, und welche Daten liegen in diesen Speicherbereichen?
Stack und Heap:

- Stack enthält:
    - Verwaltungsinformationen
    - Parameter
    - und Variablen der Methoden.

- Heap enthällt alle durch new erzeugten Objekte.

Wozu dient Garbage Collection und wie erledigt ein Garbage Collector seine Aufgabe?
Garbage Collector: automatische Freigabe des Speicherplatzes, der von nicht mehr zugreifbaren Objekten belegt ist.

Vorgehensweise: Sucht nach zugreifbaren Objekten. Markiert diese. Entfernt alle nichtmarkierten Objekte (= die, auf welche er nicht zugreifen konnte). Zuerst werden Parameter und lokale Variablen in Stackeinträgen durchsucht. Parameter und Variablen, die Objekte enthalten, verweisen auf die im Heap von den Objekten belegten Speicherbereiche. Von dort sucht der Garbage Collector in allen Variablen der Objekte weiter, bis alle auffindbaren Objekte markiert sind.

Wie kann man beim Programmieren den Garbage Collector unterstützen?
In dem man die Inhalte aller nicht mehr benötigten Variablen auf null setzt. Damit können die von den Objekten in den Variablen belegten Speicherzellen möglicherweise freigegeben werden, wenn diese Objekte nicht auch noch in anderen Variablen liegen.

Welche Fallen bestehen bei Garbage Collection?

- gefühlt immer zum falschen Zeitpunkt (effizient, aber manchmal merkbar längere Antwortzeiten)
- Speicherverwaltung machtlos gegen schlechte Algorithmen. Wenn benötigter Speicher den verfügbaren Speicher übersteigt.

- Nicht sinnvoll, wenn die Speicherbereinigung zu einer festgelegten Zeit erfolgen soll, da GC automatisch ausgeführt wird. (z.B. sicherheitskritische Systeme)
- Auf-null-setzen kann aufwendig sein wenn man keinen direkten Zugriff darauf hat. In diesen Fällen nimmt man den höheren Speicherverbrauch in Kauf.

Wie kann man beim Programmieren die Garbage Collection beeinflussen?

- Stackgröße erhöhen: über Option -Xssn1m den Stack auf 1 Megabyte setzen.
- Heapgröße bestimmen: -Xmsn6m min-größe: 6m, -Xmxn66m max-größe: 66m.
- GC explizit aufrufen:

```
Runtime r = Runtime.getRuntime();
r.gc();
```

  Dadurch steigt die Wahrscheinlichkeit, dass GC zu einem unerwünschtem Zeitpunkt nicht passiert.
- Parameter der Gargabe Collection einstellen (kompliziert)
- Mit der Methode finalize kann man Ressourcen freigeben, bevor GC den Speicher freigibt.
- Free Lists umgeht garbage collection.

Bei einem StackOverflowError kann man den Stack zu vergrößern versuchen. Warum hat man damit nur selten Erfolg?

Weil die Ursache für Stack Overflow meistens eine Endlosrekursion ist und der Fehler nur verzögert wird.

Wozu dient die Methode finalize, und warum wird sie nur selten verwendet?

Freigabe von Ressourcen vor der Ausführung von GC.
Durch Überschreiben von finalize kann der Speicherplatz nicht gleich freigegeben werden, weil vorher die Methode ausgeführt werden muss, und der Speicherbedarf dadurch steigen kann.

Wie verwendet man eine Free List?

Für jede Objektart wird eine Liste mit diesen Objekten angelegt. Wird ein Objekt benötigt, nimmt man das erste Objekt aus der entsprechenden Liste und initialisiert es neu. Wenn es nicht mehr benötigt wird, hängt man es wieder in die Liste ein.

Welche Arten von Streams können wir in Java unterscheiden?

- Gepufferte Streams und Ungepufferte Streams
- Character-Streams und Byte-Streams

Wozu benötigt man im Zusammenhang mit Streams die Methode flush?

Diese Methode zwingt alle gepufferten (aber noch nicht geschriebenen) Output Streams in die Datei zu schreiben.

Was kann passieren, wenn mehrfach von derselben Datei gelesen bzw. auf dieselbe Datei geschrieben wird?

Es kann zu Dateninkonsistenzen kommen.

Mehrfach auf Datei schreiben; Bei gepufferter Ausgabe wird ein interner Puffer (bestimmter Größe) immer wieder befüllt und dann in einem Schritt in die Datei geschrieben. Größe

der Blöcke = Puffergröße. Um das zu vermeiden muss unmittelbar (siehe Listing 6.1) nach **`out1.newLine() out1.flush()`** folgen und nach **`out2.newLine() out2.flush()`** folgen.
Weiters: Falls das zweite Argument im Konstruktor von FileWriter weggelassen wird, erhalten wir eine leere Datei, weil beim Öffnen der Inhalt gelöscht wird, sodass danach auch nichts mehr gelesen werden kann. (ohne Argument: Datei ggf. überschreiben; mit Argument: Inhalte ggf. anhängen)

Welche Fehler passieren leicht beim Umgang mit Dateien?

- Es wird oft auf Schließen oder Hinausschreiben vergessen
- Unzugreifbare Objekte belegen Dateien, diese Ressourcen können nicht freigegeben werden
- Falsche Zeichen-Codierung
- Lock-Dateien führen manchmal zu unerwünschten Ergebnissen
- Verwenden von PrintStream macht die Daten unleserlich für Scanner

Was ist eine Zeichen-Codierung?
Zeichen Codierungen sind Standards, die festlegen, wie Zeichen auf Bytes abgebildet werden
Beispiele: ASCII, UTF-8, UTF-16, ISO/IEC 8859 bzw Latin-P3N15

Wozu dienen Lock-Dateien?
Sie verhindern, dass mehrfach auf dieselbe Datei geschrieben oder gleichzeitig geschrieben und gelesen wird.

Was versteht man unter einer Antwortzeit?
Die Zeit, die zwischen der Eingabe von Daten (beispielsweise Drücken der Enter-Taste) und dem Erhalt eines Ergebnisses vergeht

Wodurch kann die Antwortzeit stärker als erwartet erhöht werden?

- Benötigte Ressourcen nicht verfügbar
- durch Ausführung von (versteckten) Aufgaben neben eigentlichen Aufgaben
- durch aufwendige (benutzerfreundlichere?) Präsentation von Ausgabedaten
- manchmal werden längere Laufzeiten in Kauf genommen, wenn dadurch Entwicklungszeiten niedriger sind und Wartung einfacher
- übereinanderliegende Technologieschichten führen zu längeren Antwortzeiten
- manchmal werden Programme auf effizienten Speicherverbrauch optimiert und nicht auf die Antwortzeiten

Was bedeutet Busy Waiting?
Aktives Warten auf ein Ereignis (z.b. Tastatureingabe), das durch wiederholten Abfragen des Status (z.b. der Tastatur) erreicht wird. Das Programm verbraucht ~~den Arbeitsspeicher~~ CPU Zeit, obwohl es selbst nichts tut.

Welche Ansätze gibt es, um Schäden durch versteckte Aktivitäten von Programmen gering zu halten?

- Open Source Software – Quellcode von Software wird veröffentlicht

- Zertifizierte Software – werden von seriösen Firmen untersucht
- Virenscanner - untersuchen ein System regelmäßig auf das Vorhandensein von als schädlich bekannter Software
- Sandboxing - die Software wird isoliert von anderen Prozessen ausgeführt (zB Apps unter iOS, Tabs in modernen Browsern)
- Programme **nicht** als Administrator ausführen
- (Mandatory Access Control wie z.B.: SELinux oder AppArmor)

Welche Fallen lauern typischerweise beim Rechnen mit ganzen Zahlen?
Wenn der Wertebereich der primitiven Typen zu groß ist kann es zum Überlauf kommen. Es führt zu falschen Ergebnissen.
Wenn die darzustellende Zahl zu klein ist, um vollständig dargestellt zu werden, kann es zum Unterlauf kommen.
Wenn die Zahl mehr Stellen braucht, als dafür im Programm vorgesehen.
Wenn man statt ganzen Zahlen Fließkommazahlen nimmt kommt es zu Rundungsfehlern.
Division durch 0.
Der Wertebereich der Positiven und Negativen Zahlen ist nicht vollkommen symmetrisch, daher kann die Negation einer Negativen Zahl zum Überlauf führen.

Was ist ein Überlauf oder Unterlauf?
- Überlauf: Falls die Zahl größer ist als der Wertebereich werden alle Bits, für die kein Platz ist, abgeschnitten.
- Unterlauf: Falls die (negative) Zahl kleiner ist als der Wertebereich werden alle Bits, für die kein Platz ist, abgeschnitten.

Wann müssen wir BigInteger statt int oder long einsetzen?
Wenn der Darstellungsbereich von int (32 bit) bzw long (64 bit) nicht ausreicht um eine Zahl darzustellen (oder wenn der Wertebereich der Zahl nicht abschätzbar ist).

Welche Arten von Problemen bei nicht abschätzbar großen Zahlen kann auch BigInteger nicht vermeiden?
Wenn es im Programm eine Obergrenze für die Anzahl der Stellen gibt.

Warum ist es meist keine gute Idee, ganze Zahlen durch Fließkommazahlen zu ersetzen, wenn der Wertebereich der ganzen Zahlen möglicherweise nicht ausreicht?
Beim Rechnen mit Fließkommazahlen kommt es zu Rundungsfehlern. Ganze Zahlen nimmt man, wenn man fehlerfrei rechnen muss (in Z logischerweise).

Wieso ist es auch bei ganzen Zahlen wichtig, klar zwischen equals und == zu unterscheiden?
== vergleicht Objektidentität und nicht die Gleichheit. Bei Primitiven Typen ist es ausreichend. Wenn man jedoch Referenztypen mittels == vergleicht, kommt es zu falschen Ergebnissen, da zwei Objekte gleichen Wert haben können, aber nicht Identisch sind.

Wofür verwendet man BigDecimal?
Für Festkommazahlen, da diese in Java mit Primitives nicht darstellbar sind. BigDecimal eignet sich gut für das Rechnen mit Geldbeträgen.

Welche Schwierigkeiten treten beim Rechnen mit Geldbeträgen auf?
Schwierigkeiten bereitet die Forderung, dass Summen immer genau zu stimmen haben. Wenn beispielsweise 1,00 Euro in drei gleiche Teile geteilt werden soll, so erhalten wir zwei Beträge von 0,33 Euro und einen von 0,34 Euro; drei Beträge von 0,33 Euro sind nicht erlaubt, da wir dabei nur auf eine Summe von 0,99 Euro kommen würden.

Was ist eine Auslöschung, und welche Algorithmen und Probleme sind (im Zusammenhang mit Fließkommazahlen) gut bzw. schlecht konditioniert?
Beim Rechnen mit fast gleich großen Fließkommazahlen kommt es zu Verlust der relevanten Stellen. Dies nennt man Auslöschung.
Dadurch fällt ein Rundungsfehler viel stärker ins Gewicht, als das normalerweise der Fall wäre.
Die Anzahl der sinnvollen Stellen hängt oft vom verwendeten Algorithmus ab. Ein Algorithmus kann:
Gut konditioniert sein: wenn im Endergebnis viele Stellen sinnvoll sind
Schlecht konditioniert sein: wenn nur wenige Stellen sinnvoll sind.

Wie entsteht POSITIVE_INFINITY, NEGATIVE_INFINITY und NaN?
POSITIVE_INFINITY: Überlauf einer Fließkommazahl, Division einer positiven Zahl durch 0.0
NEGATIVE_INFINITY: Unterlauf einer Fließkommazahl, Division einer negativen Zahl durch 0.0
NaN: Wenn das Ergebnis nicht definiert ist, zB Division durch 0 (Infinity/Infinity, 0/0)

Referenz: http://www.concentric.net/~Ttwang/tech/javafloat.htm

Ist 0.0 dasselbe wie -0.0? Was ergibt 0.0 == -0.0?
0.0 ist nicht das selbe wie -0.0
-0.0 kommt z.B. raus wenn eine negative Gleitkommazahl so nah bei 0 ist das sie nicht dargestellt werden kann
0.0 == -0.0 => true
2.0 / -0.0 => NEGATIVE_INFINITY
2.0 / 0.0 => POSITIVE_INFINITY

Warum sollte man Fließkommazahlen weder mittels == noch mittels equals vergleichen?
Aufgrund von Rundungsfehlern können zwei ähnliche Zahlen als nicht gleich ausgewertet werden. Meist will man Zahlen als gleich betrachten, auch wenn sie sich um einen kleinen Betrag unterscheiden.

Wie kann Absorption bei Fließkommaberechnungen zu einem Problem werden?
Wenn viele kleine Zahlen zu einer großen Zahl addiert werden kann die Rundung bewirken, dass die kleinen Zahlen ignoriert werden, da die Summe der kleinen Zahlen im Vergleich zur großen nicht vernachlässigbar ist, die einzelnen Zahlen jedoch schon.

Welche Fallen lauern im Umgang mit null?
-) Man muss Fallunterscheidungen machen, um zu verhindern, dass man eine Nachricht an den Inhalt einer Variablen schickt, der womöglich null ist. – (NullPointerException!)

-) wenn man null akzeptiert, muss klar sein, wofür null genau steht

Welche Vorteile dürfen wir uns dadurch erhoffen, dass wir die Verwendung von null auf das unbedingt nötige Ausmaß reduzieren (Beispiel)?
lesbarerer, kompakter Code

Was sind off-by-one-Fehler, und wodurch entstehen sie?
Fehler, in denen beispielsweise eine Schleife mit einem um eins zu kleinen oder zu großen Index beginnt oder um eins zu früh oder zu spät abbricht
Allgemein wird die Grenze für einen Programmteil (z.B.: Schleifen, Arrays) um 1 neben die gewünschte Grenze gesetzt.

Wie kann man off-by-one-Fehler vermeiden oder erkennen?

- Zusicherungen sollen Grenzen klar beschreiben
- Code Reviews können Fehler an Grenzen aufzeigen
- Beim Testen sollte man Randbedingungen, Sonderfälle und Grenzen überprüfen
- Statisches Verstehen der Grenzen hilft beim Aufspüren von Fehlern
- Terminationen überprüfen
- Plausibilitätsprüfungen

Wie kann man durch Ausnutzen eines schlecht überprüften Randbereichs einen Computer angreifen bzw. in ihn eindringen?
Wenn es einen Fehler gibt, durch den Zugriffe außerhalb des Speicherbereichs möglich sind, kann man gezielt solche Daten in ein Programm füttern, sodass bestimmte Teile des gerade ausgeführten Programms überschrieben werden. Statt der ursprünglich im Programm stehenden Anweisungen werden danach die vom Angreifer eingeschleusten Anweisungen ausgeführt. Auf diese Weise bekommt der Angreifer Zugang zu allem, worauf das Programm zugreifen darf.

Nennt man (Remote) Code Execution die zum Beispiel durch eine Buffer Overflow erreicht wird.
http://en.wikipedia.org/wiki/Arbitrary_code_execution

Was sind Pufferüberläufe, warum stellen sie eine große Gefahr dar, und was kann man dagegen tun?
Wenn man nicht genau überprüft, ob die eingegebenen Daten im Speicherbereich Platz haben, kann ein Angreifer entsprechend lange Daten in das Programm füttern, die dann andere Speicherbereiche, vor allem Teile des Programms überschreiben.
Durch Plausibilitätsprüfungen kann man Buffer Overflows vermeiden. Man muss sicherstellen, dass die Daten genug Platz haben.

~~Buffer/Heap Overflows sind auf Grund von AR SLund DEP tot.~~ ~~Das ich nicht lache. ASLR und DEP machen Angriffe schwieriger, aber noch lange nicht unmoeglich.~~ wenn man nur aslr und dep verwendet stimmt das, wenn man aber zb propolice, pax, buffershield, stackdefender, libsafe, pointguard, isa randomization, formatguard, cfi/xfi oder ähnliches dazunimmt wird das schon wesentlich schwerer (ich sage nicht unmöglich - siehe pwn2own 2010 - es wäre mir

jedoch was neues wenn ein buffer overflow PoC dafür existieren würde). in java hat man durch das übersetzen in bytecode (der überwacht wird) die buffer overflows in der jvm und nicht im programm selbst.
[PK deprecated lvl = 999]

Wodurch unterscheidet sich die Parallelität von der Nebenläufigkeit?
Parallelität: gleichzeitige (= parallele) Ausführung mehrerer Programme oder Programmteile auf mehreren Prozessoren, Prozessor-Kernen oder Recheneinheiten.
Nebenläufigkeit: Strukturierung eines Programms auf eine Art und Weise, dass einzelne Teile so miteinander kommunizieren, als ob sie gleichzeitig ausgeführt werden würden. Es spielt keine Rolle, ob sie tatsächlich gleichzeitig ausgeführt werden, oder ob die Gleichzeitigkeit nur simuliert wird.

Welche Unterschiede gibt es zwischen Multiprocessing und Multithreading?
Beim Multiprocessing können mehrere Prozesse gleichzeitig oder derart überlappt laufen, dass es so aussieht, als ob sie gleichzeitig laufen würden. Prozesse sind unabhängig voneinander.
Beim Multithreading kann ein Prozess mehrere Threads haben, die gleichzeitig oder derart überlappt laufen, dass es so aussieht, als ob sie gleichzeitig laufen würden. Mehrere Threads können auf die selben Variablen und Objekte zugreifen. Man muss darauf achten, dass sich die Threads nicht gegenseitig behindern.

Multiprocessing passiert auf Hardware-Ebene und Multithreading auf Software-Ebene.

Was versteht man unter dem Aufspannen eines Threads?
Aufrufen eines Threads innerhalb eines existierenden Threads (forking bei POSIX)

Was ist eine Race Condition?
Mehrere Ausführungen desselben Programms können zu unterschiedlichen Ergebnissen führen, weil der zeitliche Ablauf jeden einzelnen Threads nicht genau genug vorherbestimmt ist. Das heißt, die Ergebnisse von Berechnungen können davon abhängen, ob ein Thread schneller ist als ein anderer.

Was passiert bei der Synchronisation und wozu braucht man Synchronisation?
Bei der Synchronisation sorgt Java dafür, dass Methoden auf dieselben Objekte nicht gleichzeitig, sondern nur hintereinander ausgeführt werden können.
Wird dieselbe Methode von mehreren Threads aufgerufen, so werden die Threads in eine Warteliste gestellt und können mit der Ausführung erst fortfahren, wenn alle Threads vor ihnen die Ausführung der Methode schon beendet haben.
Man braucht Synchronisation um atomare Aktionen zu erreichen.

Warum spielen atomare Aktionen bei der nebenläufigen Programmierung eine wichtige Rolle?
Atomare Aktionen können nicht unterbrochen werden. Daher werden sie von den Parallellaufenden Threads nicht beeinflusst.

Wozu verwendet man wait und notify in Java?
Nach einem Aufruf von wait werden Ausführungen von synchronized Methoden auf dem Objekt vorübergehend möglich, aber der aktuelle Thread muss so lange warten, bis er durch die Ausführung von notify auf demselben Objekt durch einen anderen Thread wieder aufgeweckt wird.

Wodurch können sich nebenläufige Threads gegenseitig behindern (Liveness Properties)?

- Starvation: wenn ein oder mehrere Threads über eine längere Zeitspanne Ressourcen blockieren, sodass andere Threads nie zu diesen gelangen können.
- Livelock: Threads warten gegenseiting auf eine Aktion des anderen Threads, bevor sie ihre Aktion ausführen. Dabei fragen sie sich gegenseiting ab, ob die Veränderung schon durchgeführt wurde. Gegenseitiges Busy Waiting.
- Deadlock: ähnlich wie Livelock, nur befinden sich die Threads in der Warteliste und warten darauf, das jeweils andere fertig wird.

Warum sollen synchronisierte Methoden nur kurz laufen?
Zur Vermeidung von Starvation. Außerdem gehen dadurch die Vorteile der Parallelität verloren.

Welches Ziel verfolgt die strukturierte Programmierung?
Das Ziel besteht darin, Algorithmen und Programme so darzustellen, dass ihr Ablauf für einen Leser einfach zu erfassen ist. Wichtig ist auch, dass die Kontrollstrukturen uneingeschränkt miteinander kombinierbar sein müssen.

Welche Strukturen setzt man in der strukturierten Programmierung ein?

- Sequenzen (hintereinander auszuführender Anweisungen)
- Alternativen (durch ein- und mehrfache Verzweigungen)
- Wiederholungen (durch Schleifen oder Rekursion)
- ***sonst nichts!***

Was ist schlecht an Goto, Fall-through, Break, Continue, Return, Ausnahmen und Dangling-else?
Goto: dadurch wird die Ausführung abgebrochen und zu einer Stelle im Programm gesprungen
Fall-Through: Wird die Anweisungssequenz einer case-Klausel nicht mit break abgeschlossen, fällt man automatisch in den Code für die nächste case-Klausel
Break: wird oft zum vorzeitigen Ausstieg aus einer Schleife verwendet. Eine solche Verwendung ist jedoch zu vermeiden, da dadurch das Ziel eines einzigen, klar definierten Endpunktes verletzt wird.
Return: durch Return steigt man an beliebiger Stelle aus der Methode aus.
Ausnahmen: Werfen einer Ausnahme unterbricht den Kontrollfluss. Solang nur in seltenen Ausnahmefällen Ausnahmen geworfen werden, ist dagegen nichts einzuwenden. Problematisch wird es dann, wenn Ausnahmen dazu verwendet werden, den normalen Kontrollfluss zu unterbrechen (quasi als goto). Davon ist dringend abzuraten.
Dangling-else: bei verschachtelten If-Anweisungen ist einem Leser oft nicht klar, ob das else

sich auf das erste oder zweite if bezieht

Welche typischen Fallen lauern in der objektorientierten Programmierung?
Ersetzbarkeit: Schwere Fehler entstehen, wenn man Ersetzbarkeit fälschlicherweise annimmt
Zusicherungen: manchmal verwendet man Zusicherungen, um komplizierte Bedingungen für die Verwendung von Methoden festzulegen, damit die Methoden etwas einfacher zu implementieren sind. Dadurch erhöht sich die Komplexität des Programms.
Kovarianz: die Versuche, Kovarianz zu lösen führen oft zu Fallen
Cast: sind fehleranfällig
Vererbung: durch Vererbung kann es zur Verletzung der Ersetzbarkeit kommen.
Effizienz: um Effizienz zu erreichen, versucht man dynamisches Binden zu vermeiden und dadurch kommt es zu Ineffizienz.

Welche Fallen sind typisch für Java?
Klasse oder Interface: Manchmal ist es schwer zu entscheiden ob für das Programm eine Klasse oder ein Interface besser geeignet ist.
Überladen: Manchmal ist es schwer zu erkennen, welche der überladenen Methoden verwendet wird.
Überladen vs. Überschreiben: Es passiert leicht, dass man unabsichtlich beispielsweise den Typ eines Parameters ändert oder zwei Parameter vertauscht.
Ersetzbarkeit von Arrays: Wenn ein Objekt der Untertyp eines anderen Objekts ist, so ist die Arrayliste dieses Objekts auch Untertyp von der Arrayliste des Obertyps.
Raw Types: Wenn man eine Instanz einer generischen Klasse bildet, ohne einen Typ anzugeben, so wird der Raw Type verwendet. Diese Instanz akzeptiert Eingaben jeder Art (zb: Continue: wird verwendet um noch vor Beendigung einer Iteration für die nächste Iteration an den Anfang der Schleife zurückzuspringen. Seiteneffekte, die gegen Ende der Schleife passieren sollten, werden wegen der Beendigung der Iteration nicht passieren
String und Int gleichzeitig), beim Ausgeben der Werte kann es aber zu Fehlern kommen.
Pakete: In Java bilden alle Klassen in einem Verzeichnis ein Paket. Das kann jedoch zu Problemen führen, wenn man die Verzeichnisstruktur ändern oder an neue Gegebenheiten anpassen will (*nur wenn man JEdit verwendet*).
Falsche Sicherheit: Java gilt als sichere Programmiersprache, die Sicherheit wird jedoch oft überschätzt, obwohl sie nicht in allen Bereichen gegeben ist. Das führt dazu, dass oft zu wenig getestet wird und sich auf etwas verlassen wird, was nicht gegeben ist. (es gibt keine sicheren Programmiersprachen - nur sichere Programme)

Wie unterscheidet sich die defensive von der offensiven Programmierung?
Defensiv: Programmierstil, bei dem man sich nicht blind auf irgendwelche Bedingungen verlässt, sondern lieber alles mehrfach überprüft. “Vetrauen ist gut, Kontrolle ist besser.”
Offensiv: den Bedingungen wird grundsätzlich vertraut, solange es keine Probleme gibt.
“Solange sich niemand beschwert, wird es schon passen.”

In welchen Zusammenhängen ist ein defensiver Programmierstil nötig?
Beim Validieren der Daten aus externen Quellen.
Bei Zusicherungen in den Programmschnittstellen.

Warum sind Überprüfungen mancher Bedingungen (bei einem defensiven

Programmierstil) im Zusammenhang mit Zusicherungen oft schwierig bzw. verzichtbar?
Systematisch durchgeführte unnötige Überprüfungen führen auch leicht dazu, dass der Vertragspartner seine Verantwortung für die Einhaltung des Vertragsbestandteils nicht mehr ernst nimmt; schließlich werden die Bedingungen ohnehin nocheinmal überprüft.

Warum kann ohne Vertrauen keine gute Software entstehen?
Damit gute Software entsteht sollte man auf die Wiederverwendung von Programmteilen vertrauen können, auf stabilen Code und auf den Aufwand welcher in die Entwicklung eines Programmstücks gesteckt wurde. Wenn ein Code sich bewährt hat und stabil ist kann man darauf vertrauen, dass eine gute Software daraus entwickelt wird.
Vor allem ist wichtig, dass die Zusicherungen von allen Programmteilen eingehalten werden, da diese sonst (unnötig) mehrfach kontrolliert werden, was zu einem Verlust von Laufzeit führt.

Welche Aspekte sind zur Gewinnung von Vertrauen in Programmcode wichtig?
Schnittstellen: Die für die Benutzung eines Programmstücks notwendigen Informationen müssen klar und deutlich bekanntgegeben werden
Verständlichkeit: logisch strukturierter und gut lesbarer Code wird positiv auffallen
Entwicklungprozesse: Es muss transparent sein, wie bei der Konstruktion jeden Programmteils vorgegangen wird
Wartung: wenn das Programmstück nicht gewartet wird, wird man nicht mehr darauf vertrauen
Standardkonformität: Wenn man Standards ignoriert, besteht Verdacht, dass man in diesem Bereich einfach nicht genug Wissen hat, um ein Problem effektiv lösen zu können

Wie kann Misstrauen im Team zum Scheitern von Softwareprojekten führen?
Misstrauen innerhalb eines Teams führt zu

- Eigenbrötelei und mangelnder Kommunikation im Team,
- Mehrgleisigkeiten, weil Lösungen mehrfach entwickelt werden statt bereits fertige Lösungen zu verwenden,
- viel zusätzlichem Code für eigentlich sinnlose Überprüfungen,
- hoher Fehleranfälligkeit durch widersprüchliche überprüfte Bedingungen und inkonsistente Mehrfachlösungen,
- hohem Ressourcenverbrauch sowohl in der Entwicklung als auch in der Programmausführung
- und schließlich sehr oft zum Scheitern eines Projekts.

Wozu dienen Teamregeln?
Sie sollen einen einheitlichen Programmierstil erzwingen (eher fördern) und damit dem Team diesbezüglich eine eigene Identität verschaffen und das gegenseitige Vertrauen fördern.

Warum ist es notwendig, ein Gespür für den Wahrheitsgehalt von Mythen zu entwickeln?
Damit man sich eine eigene Meinung bilden kann und nicht von den Standpunkten Dritter abhängig ist.

Zählen Sie typische Mythen im Bereich der Programmierparadigmen und von Java auf

und analysieren Sie deren Wahrheitsgehalt.
Imperative Programmierung effizient: Effizienz kommt hauptsächlich von der Wahl der Algorithmen
Auf Objekte kommt es an: Bei einem kleinen Programm zahlt sich der Aufwand nicht aus.
Objektorientiertheit längst out: tatsächlich bildet OO die Grundlage für neue Konzepte von denen behauptet wird, dass sie OO verdrängen.
Funktionale Programmierung für Freaks: In einem funktionalen Stil kann man die meisten Algorithmen viel einfacher und kürzer ausdrücken als in einem imperativen Stil. (einfacher ist relativ - mathematische konzepte/prinzipien lassen sich einfacher implementieren weil man sich mehr darauf fokusiert was man tut und nicht wie man etwas tut)
Typüberprüfungen schützen vor Fehlern: davon sind nur eher einfache Fehler betroffen
Zukunft ist dynamisch: Die Deklaration von Typen kann aber die Lesbarkeit erhöhen und dadurch Fehler reduzieren (widerspricht das nicht dem vorigen mythos? ^^)
Technologien sind entscheidend: Die Fähigkeit der Programmierer ist entscheidend!

Mythen in Java:
Portablität
Sichere Sprache
Fallen beseitigt
Internet-Sprache Java ist nicht Javascript (kommt drauf an was man unter “Internet-Sprache” versteht; mit Java kann man durchaus Web Applications entwickeln die serverseitig laufen)
C viel effizienter als Java: bei optimiertem Code nur mehr rund 2 mal so schnell; hängt vom konkreten Anwendungsfall ab und lässt sich nicht allgemein sagen
Java ist ein alter Dinosaurier: stimmt. Man vergleiche zb mit Scala xD <= das **versucht** Java durch hunderte Libraries auszugleichen :X
Jede Programmiersprache, die länger im Gebrauch ist (wie Java) neigt dazu, immer größer zu werden, das ebenfalls hohe alter ergibt sich durch die lange Benutzung von alleine.