HotSpot How To

Running tests and pushing changes

Read about the Submit repo here.

Before pushing

In order to push a change to the HotSpot source code you need to complete all the steps in the list below. This is done in order to ensure code quality and reduce the risk of introducing bugs into the code base.

1. You must be a Committer in the JDK project
2. You need a non-JEP JBS issue for tracking
3. Your change must have been available for review at least 24 hours to accommodate for all time zones
4. Your change must have been approved by two Committers out of which at least one is also a Reviewer
5. Your change must have passed through the hs tier 1 testing provided by the submit-hs repository with zero failures
6. You must run all relevant testing to make sure your actual change is working
7. You must be available the next few hours, and the next day and ready to follow up with any fix needed in case your change causes problems in later tiers

There is a notion of ***trivial changes*** that can be pushed sooner than 24 hours. It should be clearly stated in the review mail that the intention is to push as a trivial change. How to actually define "trivial" is decided on a case-by-case basis but in general it would be things like fixing a comment, or moving code without changing it. Backing out a change is also considered trivial as the change itself in that case is generated by mercurial.

Relevant testing

Please note that the submit repository will only run a set of smoke-tests to ensure your change compiles and runs on a variety of platforms. It will not do any targeted testing on the particular code you have changed. Running through the submit repository is only the minimum requirement. You must also make sure your change works as expected before pushing using targeted testing. Consider writing a few JTREG tests for your change, or some unit tests using the GTest framework. Including the new tests (in the right places) with your push to the submit repository will ensure your tests will be run as part of your testing on all platforms and in the future. Look for tier1 in test/hotspot/jtreg/TEST.groups to see which tests and directories that are included in the submit repo testing.

The push

Pushing a change is fairly straight forward. Make sure your commit has a proper description. The JBS bug id and the Rewiewed-by lines are mandatory.

```
8197844: JVMTI GetLoadedClasses should use the Access API
Summary: Make sure the holder of a class loader is accessed during iteration of CLDG
Reviewed-by: eosterlund, rkennke
```

Always make sure there are no new changes in the repository you are pushing to before pushing

`hg in` should return an empty set of changes when you push. Always verify this the last thing you do! If there are new changes in there you must pull these changes and rebase your patch on top of these new changes. We use rebase to avoid merge changesets as we want to keep the mercurial history as linear as possible. This makes it easier to do binary searches through the history when trying to determine the cause of some changed behavior for instance (read debugging).

```
hg pull -u --rebase
```

Look through the new changes before you push your change. Did something recently pushed conflict with your change? There may be a need to rerun some testing or at least recompile before pushing.

When you feel confident enough, push using **`hg push`**.

After pushing

As noted in the list above, you are expected to be around after having pushed a change in case there are any issues with it. A change that causes failures in later tiers may be backed out if a fix can not be provided fast enough, or if the developer is not responsive when noticed about the failure. Note that #7 above should be interpreted as "it is a really bad idea to push a change the last thing you do before bedtime, or the day before going on vacation".

Test analysis and bug handling

A critical failure is a test failure that has a high impact on the daily work for developers.

This can mean a few different things:

- **Any failure in tier 1 is considered critical.** A single failure, no matter how isolated or trivial, will cause the entire tier to signal failure. As long as there are tier 1 failures, every developer that pushes a change must dig into the results to see what caused the failure. Human nature dictates that after a while one will assume that the failure is the same as it was in the previous run, and at that point tier 1 testing is useless. Unless the results are checked every single time we will not notice when a new failure appears.
- **Any failure that is blocking integration is considered critical.** Integration blockers cause delays in propagation of other changes. People outside of the HotSpot team might be waiting for a change to propagate to master. The performance team needs HotSpot changes to get into promoted builds on a regular basis. In short, integration blockers stop others from doing their work.
- **Any failure that causes multiple unrelated tests to fail is considered critical.** Bugs with a large damage area will hide other bugs and make it harder to fix those other bugs once they are found. Tests that cause massive failures in the nightly will seriously reduce our test coverage.

Please note that there is a significant difference between high priority and high urgency. High priority issues like vulnerabilities for example will always have higher priority than regular bug fixing. Critical issues are urgent, but do not always have high priority. It could be a trivial bug in a test that has no particular significance, a test that nobody cares wether it is fixed or not. But, if it causes failures in tier 1 it is urgent to fix it. Not high priority but still urgent. It is most likely urgent enough to take 20 minutes away from fixing a vulnerability.

Handling a critical failure

It is required that anyone who pushes a change monitors the tier 1 testing and the following nightly, or designates this requirement to someone who can also fix possible regressions. It is considered exceptionally bad engineering to push a change at the end of the day, or on the last day before going on vacation or otherwise becoming unavailable. Critical failures must be handled with high urgency, thus the engineer who caused the failure is best suited to do so.

If a failure in **tier 1** is identified and can be fixed within **two - three hours**, this is the preferred way to go. If a fix is not expected within that time the broken change **must be backed out**.

Integration blockers must be fixed before the next integration per definition. Usually this means that a fix should be available **within a few days**.

Bugs that causes unrelated tests to fail in the nightly should be fixed **before the next nightly**. If a fix can not be delivered within **two days** the causing change **must be backed out**.

Backing a change out is easy. Anyone can do it.

There can never be any ownership involved when it comes to backing out a change. It doesn't matter who made the change or why it fails, if it needs to be backed out it should be backed out. Whoever is on site first should do it. Usually there is some synchronization before doing the actual work to avoid duplicate work.

Use mercurial to create a backout changeset:

```
hg backout -r <revision_id>
```

A backout is considered a trivial change so a single Reviewer is enough and you can push immediately when reviewed. See How to backout a change for details on bug management in JBS related to backing out changes.

The nightly HotSpot testing starts at 8 pm PT each weeknight (Mon - Fri). To clarify what this means here are a few relevant timezones. The nightly snapshot in this example would be referred to as the Wednesday snapshot.

AEST	Thursday May 12, 1:00 pm
IST	Thursday May 12, 9:30 am
CET	Thursday May 12, 5:00 am
UTC	Thursday May 12, 4:00 am
EST	Wednesday May 11, 11:00 pm
PST	Wednesday May 11, 8:00 pm

If a new failure is found a bug should be filed in JBS. Try to make the bugs as complete as possible to make it easier to triage and investigate the bug. It's not really the reporter's responsibility to set a correct priority, but a good guess is always better than a default value. To help with setting the right priority consider things like how the bug impacts the product and our testing, how likely is it that the bug triggers, and how difficult is it to work around the bug if it is not fixed soon. To find out which component to use for different bugs look at Hotspot JBS components. Once a bug has been filed component triage should verify that labels, priority, affected version, and descriptions are correct and set the fix version.

A few things to keep in mind when filing a new bug

- **Before filing a bug**, verify that there isn't already a bug filed for this issue.
- **Any bug that is blocking integration should be labeled 'integration_blocker'.**
  - Any bug that is present in the tested repository and is not already present upstream is blocking integration. Basically all new failures are integration blockers and should be labeled as such. For more information see Integration and Integration Blockers.
- **Add other relevant labels like 'intermittent', 'regression', 'testbug'... etc.**
- **Set affects version.**
- **In the description, always include** (if present):
  - full name of the failing test(s)
  - error messages
  - assert messages
  - stack trace
  - command line information
  - relevant information from the logs
- **Always file separate bugs for different issues.**
  - If two crashes looks related but not similar enough to for sure be the same, it is easier to close a bug as a duplicate than it is to extract the relevant info from a bug to create a new one later.

All new failures are considered integration blockers

If a failure is determined to be due to a broken test and the test is not expected to be fixed before the next nightly, it is OK to **quarantine** the test to be able to get a "clean" nightly. The JBS issue used to quarantine the test should be a sub-task to the original bug and the integration_blocker label is moved to the sub-task. This way, when the test has been quarantined, the bug is no longer a blocker.

(C) Compiler
(G) GC
(R) Runtime

- hotspot/
  - cpu (C+R) - will have to be decided on a case by case basis
  - os (R)
  - os_cpu (C)
  - share/
    - adlc (C)
    - aot (C)
    - asm (R)
    - c1 (C)
    - ci (C)
    - classfile (R)
    - code (C)
    - compiler (C)
    - gc (G)
    - interpreter (R)
    - jvmci (C)
    - libadt (C)
    - logging (R)
    - memory (R+G)
    - oops (R)
    - opto (C)
    - precompiled (R)
    - prims (R)
    - runtime (R)
    - services (R)
    - trace (R)
    - utilities (R)

The HotSpot team owns two components in JBS: hotspot and core-svc.

Please note that all issues in the hotspot component are required to have a subcomponent set.

Team	JBS component	JBS query
Compiler	hotspot / compiler	`component = hotspot AND Subcomponent = compiler`
GC	hotspot / gc	`component = hotspot AND Subcomponent = gc`
Runtime	hotspot / runtime hotspot / jfr	`component = hotspot AND Subcomponent in (runtime, jfr)`
Serviceability	hotspot / jvmti hotspot / svc core-svc core-svc / debugger core-svc / tools core-svc / java.lang. instrument tools / hprof	`( (component = hotspot AND Subcomponent in (jvmti, svc)) OR (component = core-svc AND (Subcomponent is EMPTY OR Subcomponent in (debugger, tools, java.lang.instrument))) OR (component = tools AND Subcomponent = hprof) )`
Monitor and Management	hotspot / svc-agent core-svc / java.lang. management core-svc / javax. management	`( (component = hotspot AND Subcomponent = svc-agent) OR (component = core-svc AND (Subcomponent in (java.lang.management, javax.management) )`
Test	hotspot / test	`component = hotspot AND Subcomponent = test`

The **`integration_blocker`** label should be used to indicate that a bug is present in a repository but not present upstreams. When filing a new bug found in hotspot testing, the default should be to add the integration_blocker label. If it is verified (when filing or later) that the new failure is already present upstreams from the repository where it was found the label should be removed and an explicit comment must be added to the bug explaining why the bug is not an integration blocker. A failure seen in a project repo is considered escaped when found in `jdk/hs`. A failure seen in `jdk/hs` is considered escaped if seen in `jdk/jdk` or if older failures are found from a date before the last integration to `jdk/jdk`.

If an integration blocker escapes from a project repository it will become an integration blocker in `jdk/hs`.

The **`testbug`** label should be used for bugs in tests and test infrastructure. There is no fundamental difference between a test bug and a product bug in the eyes of the gatekeeper. Both are considered integration blockers and both should be handled asap when appearing in the testing.

When a change is identified that causes a regression and the best way to handle it is to back out the change, anyone can do so. It will still go through the standard code review process, but is considered a **trivial change** and thus it requires only one Reviewer and will avoid the 24h code review window. The idea here is to save time by not having a broken change hindering others. There is also the rationale that the change itself is automatically created by hg, and reviewed by the person who is performing the backout, so only one additional reviewer is required.

There are two parts to this task, how to do the bookkeeping in JBS, and how to do the actual backout in mercurial.

How to work with JBS when a change is backed out

1. Close the original JBS issue and mark it with "Fix Failed", which is an alternative on the "Verify" action.
2. If the intention is to fix the change and submit it again, create a redo-issue to track that the work still needs to be done. Clone the original JBS issue and use prefix [REDO] on the summary.
    - Make sure relevant information is brought to the clone.
3. Create a backout-issue:
    - **Alternative 1 -** a regression is identified directly. Create a Sub-Task to the redo-issue with the same summary, but prefix with [BACKOUT].
    - **Alternative 2 -** an investigation issue is created, and during the investigation backing out the change is identified as the best solution.
        a. Use the investigation issue for the backout.
        b. Change summary to the same as the issue to back out and prefix with [BACKOUT].
        c. Link the redo-issue and the backout-issue.

- **Alternative 3 -** no redo issue was created. Create a backout-issue with the same summary, but prefix with [BACKOUT].
    a. Link the backout-issue and the original issue.

- Remember that comments are not brought over.
- Quarantine and exclude labels will continue to point to the original bug (unless updated at back out). This is accepted since there is a clone link to follow.

How to work with mercurial when a change is backed out

In order to backout a change, the `hg backout` command is recommended, which essentially applies the anti delta of the change. Make sure you perform the backout in the most upstream repostiory the change has escaped to.

```
hg backout [OPTION]... [-r] REV

reverse effect of earlier changeset

    Prepare a new changeset with the effect of REV undone in the current
    working directory.

    If REV is the parent of the working directory, then this new changeset is
    committed automatically. Otherwise, hg needs to merge the changes and the
    merged result is left uncommitted.
```

To remove noise from our testing, tests that are expected to fail are quarantined or excluded. A quarantined test is removed from standard runs, but run in a separate job. Excluded tests are not run.

Test quarantines are done in the file `ProblemList.txt` while exclusions shall be done in the source code using `@ignore`. This enables adhoc runs on local machines, it makes it possible to go back in time and get expected results, and (for good and bad) by being part of the standard code review process the visibility increases. For each quarantine and exclude, there should of course be a corresponding bug, which is referenced in the quarantine.

Quarantine jtreg tests

Use `ProblemList.txt` and exclude the test in the known issues ignore list.

Example where MyTest.java is excluded on windows, tracked by bug JDK-4711:

```
sun.tools.jcmd.MyTest.java                        4711   windows-all
```

Exclude jtreg tests

Use `@Ignore` with a bug reference in the test case, to prevent the test from being run.

Example where MyTest.java is excluded, tracked by bug JDK-4711:

```
/**
 *  @test
 *  @ignore 4711
 *
```

`@ignore` should always be placed directly before the first `@run` line in the test.

Dealing with JBS bugs for test exclusion

With the quarantine/exclude mechanism in the source code, a check-in into the repository is needed, which means a unique JBS issue and a code review is needed. This is a good thing since it makes the test problems visible.

- **Code review:** Since it is a trivial change, it only needs a review from one official Reviewer and don't need to wait 24 hour before commit.
- **JBS issue:** A JBS issue is created for the bug. Create a subtask for the test exclusion checkin.

Removing tests should not be the standard thing to do. A failing test is expected to be a regression, and should be handled promptly with high priority. Preferably it is fixed, possibly it is backed out.

Triage quarantined issues

After a failure is handled by a test quarantine, the JBS issue for the remaining product/test bug should be re-triaged and possibly given a new priority. This should be handled by the standard triage process. When the triage team does re-triage the issue, it should not only consider the impact of the bug itself, but also the outage on the testing the quarantine of the test results in.

Who looks at quarantine batches?

The gatekeeper does not review the quarantine batches as part of daily work, the point is to reduce the load for the gatekeeper. It is the responsibility of the team who owns the bugs to keep an eye at this in general. It is for a team to be aware of the risk of quarantining tests and fix it accordingly. The batches are still run to verify the state of the quarantine batches and supply statistics on the results. It is a good idea to make sure the state of the batches don't get worse.

Any rule, any process has exceptions. And in some cases it is better to live with a potential failure and integrate anyway. It should of course be used in rare cases only, for example when it is more important to continue to run the test, and thus at the same time accept the failure. A typical example is an intermittent failure like an intermittent crash (say once a month) in a large system test, or if more info is needed, possibly diagnostics are added to the test, and the risk of not running it is high.

An exception like this requires that someone is actively working on the issue.

Label the bug with **hs-nightly-quarantine-exception**.

There are a few ways to figure out if a problem has already escaped a repository:

- The easiest way is of course if you know which change that caused the problem. Just have a look at the hg log of the upstream repository to see if the change is in there.
- Searching JBS can give information as well. See if there is more than one bug filed for the issue. If it escaped it is likely that someone else has found it and maybe filed a new bug.

A bug that has escaped to an upstream repository should be fixed in the upstream repository

If a bug has escaped from a project repository into `jdk/hs`, the fix/backout/quarantine should be done in `jdk/hs` to avoid blocking integration for other Hotspot changes. If a bug escapes `jdk/hs` into `jdk/jdk` the fix must go into `jdk/jdk`.

Integration

Most HotSpot development work is done in `jdk/hs`. Larger projects are recommended to use a project repository which is synced with `jdk/hs` on a regular basis.

When is a change integrated?

Changes in `jdk/hs` are bulk integrated to `jdk/jdk` once a week. A change in HotSpot will go through these steps on the way to `jdk/jdk`:

1. After proper testing HotSpot developers push changes to the HotSpot repo.
   - Pushes are direct and done using **hg push**
2. When a push is done **Continuous Integration (CI)** will automatically start a set of test jobs on the HotSpot repo.
   - These tests include hs-tier1, hs-tier2, jdk-tier1, jdk-tier2, jdk-tier3, and builds-tier1.
   - It is obviously a good idea to have run the relevant tests before pushing to ensure that no failures happen in CI.
   - If a CI job is already running, the next job will start once the current is done. If several pushes have been made in this time, all will be tested in the same CI job.
   - CI failures are considered **critical failures** and must be dealt with immediately.
3. Every weeknight (Mon - Fri) a **nightly test** job is started at 8 pm PT.
   - The nightly testing runs hs-tier3, hs-tier4, and hs-tier5.
   - New failures that are encountered in the nightly testing are always labeled as **integration_blockers**.
4. Each Friday the changes included in the most recent clean nightly are merged with the most recent clean CI build of master. This merge is then sent to **PIT** (Product Integration Testing).
   - A clean nightly is one where no integration blockers were open when the nightly started, and no new ones were found in the same.
   - If all integration blockers in a nightly snapshot have been fixed before the PIT is integrated, that snapshot can be chosen provided the extra fixes are included in the integration.
5. Once the PIT result has been analysed (usually early the next week), and no integration blockers were found, the merge is **pushed to master** (using **hg push**)

All in all this means that if you need a change to be in master at a specific date, you need to make sure your change is pushed to `jdk/hs` in time to go through the PIT and be pushed to master before the actual date.

What is an integration blocker?

An integration blocker is **a bug that is present in a repository, but has not yet spread to the upstream repository**. We want to make sure not to spread problems, so if there are new failures in a repository, hold upstream integration.

There is a JBS label that is used to help keep track of bugs that are blocking integration, integration_blocker.

Never integrate if there are open integration blockers

Use the JBS filter Hotspot Integration blockers to find out if there are any blockers.

If a main bug is targeted to a release and the fix is pushed to a different release, then a backport bug is automatically created. Usually this is a "good thing", e.g., when you are really backporting a fix to an earlier release, but not always... If the main bug is targeted to a later release (due to schedule planning), but someone finds the time to fix that bug in the current release, then the bug should be retargeted to the current release before pushing the fix. However, sometimes we forget to do that.

Here is how to fix that:

In this example a fix was pushed to JDK 10 (a.k.a. the current release) while the JBS bug was targeted to JDK 11 (a.k.a. a future release).

1. **Reopen the *backport* bug that was created automatically**
   - Use a comment like the following (in the reopen dialog):

     > Fix was pushed while main bug was targeted to '11'. Reset the main bug to fixed in '10', reset this bug to fix in '11' and closed as 'Not An Issue' since JDK 11 will automatically get this fix from JDK 10.

   - Change the 'Fix Version/s' from '10' to '11'.
   - Close the *backport* bug as "Not an Issue".
2. **Clean up the main bug**
   - Copy the open push notification comment from the *backport* bug to the *main* bug, e.g.:

     > HG Updates added a comment - 6 hours ago
     > URL: http://hg.openjdk.java.net/jdk/jdk/rev/8db54e2c453b
     > User: clanger
     > Date: 2017-12-11 07:25:20 +0000

   - Add a comment like the following to the *main* bug:

     > Fix was pushed while main bug was targeted to '11'. Reset the main bug to fixed in '10' and copied the hgupdater entry here.

   - Reset the *main* bug 'Fix Version/s' from '11' to '10'.
   - Resolve the *main* bug as "Fixed" in build "team" or in build "master" depending on where the fix was pushed. Pushes to 'jdk/jdk' are fixed in build "master" and pushes to other repos are fixed in build "team".

There are several ways to find out if a change has been integrated to `jdk/jdk` or not. The easiest way is probably to look at the mercurial log in `jdk/jdk` and see if your change is there:

```
hg log | grep 1234567
```

Or, if you don't have a recent clone of `jdk/jdk` available, look at `http://hg.openjdk.java.net/jdk/jdk/`

This is an example of how to read the mercurial graph that is available at http://hg.openjdk.java.net/jdk/jdk/graph/

> In this particular example we ask ourself the question, was JDK-8186027 included in the promoted build JDK 10 b35?

We start by locating the change **JDK-8186027** (**1**). As we can see in the graph this change was introduced at the beginning of a blue line. Following this blue line upwards it goes cyan for a few changes and then merges into a yellow line at the merge after **JDK-8170244** (**2**). This yellow line continues all the way to the top. On the right side there is a green line which merges into the yellow line a few changes from the top (**3**). As we can see here the tag `jdk-10+35` was added to the green line before merging with the yellow. This means that changes in the yellow line was not covered by the tag.

Following the green line a few pages down we get to **JDK-8192885** (**4**), which is the first change in the green line, and we can see where this line was branched out from the yellow line to the left. Since there were no other changes along the green line, JDK-8192885 was the last change to get included in b35. All the changes in the yellow line below the branch was also included. No changes in the yellow line above the branch was included.

No changes in any of the other lines at the bottom of the picture was included in b35 either as they all merges into the yellow line going up the graph - this takes us back to the blue line where JDK-8186027 was pushed. This blue line is the result of a few merges of the cyan, blue, and magenta lines that are between the green and the yellow line at the bottom of the graph. Since the blue line merges into the yellow line far above (**4**), JDK-8186027 is not included in b35.